

# C Language Guide and Reference

Programming Family



Personal  
Computer  
Software



IBM RT PC Advanced Interactive Executive Operating System Version 2.1

# C Language Guide and Reference

Programming Family

**IBM**  
Personal  
Computer  
Software

---

## **First Edition (January 1987)**

This edition applies to Version 2.1 of IBM RT PC AIX Operating System licensed program, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

**International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.**

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer or your IBM marketing representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1985, 1987  
© Copyright INTERACTIVE Systems Corporation 1984  
© Copyright AT&T Technologies 1984

---

## About This Book

This *C Language Guide and Reference* describes the C programming language as implemented in UNIX<sup>1</sup> System V—Release 2. Appendix A, “IBM RT PC C Language” explains how the IBM RT Personal Computer<sup>2</sup> C Language compiler implements the C language.

### Who Should Read This Book

This book is intended for persons with some knowledge of general programming concepts and some experience in writing programs.

### How to Use This Book

This book contains guide and reference sections. If you are not familiar with programming in the C language, read “Part 1. Guide.” The guide defines the terminology used to describe the C language and describes the general structure of C language programs. After reading the guide, use “Part 2. Reference” to help you write your C language programs.

If you have written programs in the C language, you can refer to the sections in “Part 2. Reference” to gain a more complete understanding of the syntax and semantics of each C language construct. For a description of how the IBM RT PC C Language compiler implements the C language, see Appendix A, “IBM RT PC C Language.”

### What is In This Book

This book contains the following information:

- A guide to writing C language programs
- A reference to C language constructions
- An overview of the C standard library
- An overview of compiling, linking, and running programs

---

<sup>1</sup> UNIX was developed and licensed by AT&T. It is a registered trademark of AT&T in the United States of America and other countries.

<sup>2</sup> RT, RT PC, and RT Personal Computer are trademarks of International Business Machines Corporation.

- 
- Many examples of C language code interspersed throughout the text and an advanced example program located in an appendix
  - A list of IBM RT PC C Language implementation characteristics
  - A list of portability considerations
  - A list of common errors in C programming
  - A list of AIX<sup>3</sup> Operating System utilities that can aid C language programmers in developing and maintaining programs
  - A list of ASCII character codes
  - A glossary of C language terms
  - An index.

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

## **What is Not In This Book**

This book does not contain the following information:

- Instructions on how to use the IBM RT PC
- Instructions on how to develop a C program using an editor
- Descriptions of every C standard library call and subroutine
- Explanations of how to use all the C language programming utilities (debugging and formatting programs)
- Explanations of the IBM RT PC C Language compiler error messages.

---

<sup>3</sup> AIX and Advanced Interactive Executive are trademarks of International Business Machines Corporation.

---

## Highlighting

This book uses two sets of highlighting conventions, highlighting within text and highlighting within syntax diagrams.

### Text

Type styles appearing within the text show the following:

Type Style	Description
Monospace	Examples appear in monospace type.
<b>Bold</b>	Commands, messages, and keywords appear in <b>bold</b> type.
<b><i>Bold Italic</i></b>	New terms appear in <b><i>bold italic</i></b> type.

### Syntax Diagrams

Syntax diagrams appear throughout the reference section and on the poster that accompanies this book. Each diagram describes part of the C language. To accurately interpret syntax diagrams, read the diagrams from left to right.

Type styles appearing in syntax diagrams show the following:

Type Style	Description
<i>Italic</i>	The title of each diagram and parts of the language that are described by other diagrams appear in <i>italic</i> type. Look for the matching diagram title.
<b>Blue Monospace</b>	Letters, numbers, and symbols that you must type into a program appear in <b>blue monospace</b> type.
Roman	Comments about the diagrams and informal descriptions appear in Roman type.

A box that contains several items shows that you must choose one of the enclosed items. A box that appears inside a repetition loop shows that you can choose one or more of the enclosed items. The following diagram shows a box inside a repetition loop:

octal  
constant



The preceding diagram shows that the leftmost digit of an octal constant must be 0 (zero). Other octal constant digits can be any of the base 8 numerals 0 (zero) through 7.

Permitted as an Octal Constant	Not Permitted as an Octal Constant
0	0254c3
07	9
03145037	0820951
0160	092

## Related Information

The following books contain information about, or related to, C language programming:

- *IBM RT PC Bibliography and Master Index* provides brief descriptive overviews of the books and tutorial program that support the IBM RT PC hardware and the AIX Operating System. In addition, this book contains an index to the RT PC and AIX Operating System library.
- *IBM RT PC Using the AIX Operating System* describes using the AIX Operating System commands, working with file systems, and developing shell procedures.
- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands. Detailed descriptions of the C language compiler command and of the programming utilities available through the AIX Operating System are included in this book.
- *IBM RT PC AIX Operating System Technical Reference* describes the system calls and subroutines that a C programmer uses to write programs for the AIX Operating System.

---

This book also includes information about the AIX file system, special files, file formats, GSL subroutines, and writing device drivers. (Available optionally)

- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)
- *IBM RT PC Assembler Language Reference* describes the IBM RT PC Assembler Language and the 032 Microprocessor and includes descriptions of syntax and semantics, machine instructions, and pseudo-operations. This book also shows how to link and run Assembler Language programs, including linking to programs written in C language. (Available optionally)
- *IBM RT PC INed* provides guide and reference information for using the INed program to create and revise files.
- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.

See *IBM RT PC Bibliography and Master Index* for order numbers of IBM RT PC publications and diskettes.

## Ordering Additional Copies of This Book

To order additional copies of this publication, use either of the following sources:

- To order from your IBM representative, use Order Number SBOF-0134.
- To order from your IBM dealer, use Part Number 79X3859.

A binder, the *C Language Guide and Reference* manual, and the *C Language Syntax Summary* poster are included with the order. For information on ordering the binder, the manual, and/or the poster separately, contact your IBM representative or your IBM dealer.



---

# Contents

## Part 1. Guide

<b>Chapter 1. The Structure and Format of C Language Programs</b> .....	<b>1-1</b>
About This Chapter .....	1-3
Program Structure .....	1-4
Program Format .....	1-6
 <b>Chapter 2. Defining Data</b> .....	 <b>2-1</b>
About This Chapter .....	2-3
Defining Constants .....	2-4
Defining Variables .....	2-6
 <b>Chapter 3. Creating Expressions and Statements</b> .....	 <b>3-1</b>
About This Chapter .....	3-3
Using Operators to Create Expressions .....	3-4
Forming C Language Statements .....	3-6
Forming C Preprocessor Statements .....	3-14
 <b>Chapter 4. Creating and Using Functions</b> .....	 <b>4-1</b>
About This Chapter .....	4-3
Creating Functions .....	4-4
Calling Functions and Passing Parameters .....	4-6
Using the C Standard Library Functions .....	4-8
 <b>Chapter 5. Compiling, Linking, and Running a Program</b> .....	 <b>5-1</b>
About This Chapter .....	5-3
Compiling and Linking .....	5-4
Running .....	5-7
 <b>Chapter 6. Debugging a Program</b> .....	 <b>6-1</b>
About This Chapter .....	6-3
Understanding Error Messages .....	6-4
Using System Utilities to Detect Errors .....	6-5

---

## Part 2. Reference

<b>Chapter 7. Comments, Identifiers, and Reserved Words</b> .....	<b>7-1</b>
About This Chapter .....	7-3
Comments .....	7-4
Identifiers .....	7-6
C Language Reserved Words .....	7-8
 <b>Chapter 8. Data Definitions and Storage Classes</b> .....	 <b>8-1</b>
About This Chapter .....	8-3
Internal Data Definitions .....	8-4
External Data Definitions .....	8-6
Declarators .....	8-8
Initializers .....	8-12
<b>auto</b> Storage Class .....	8-14
<b>extern</b> Storage Class .....	8-19
<b>register</b> Storage Class .....	8-24
<b>static</b> Storage Class .....	8-27
 <b>Chapter 9. Constants</b> .....	 <b>9-1</b>
About This Chapter .....	9-3
Character .....	9-4
Decimal .....	9-6
Enumeration .....	9-8
Escape Sequence .....	9-10
Float .....	9-12
Hexadecimal .....	9-15
Integer .....	9-17
Long .....	9-18
Octal .....	9-19
String .....	9-21
 <b>Chapter 10. Primary Data Types</b> .....	 <b>10-1</b>
About This Chapter .....	10-3
Characters .....	10-4
Floating-Point Variables .....	10-6
Integers .....	10-8
<b>void</b> Type .....	10-10

<b>Chapter 11. Complex Data Types</b>	<b>11-1</b>
About This Chapter	11-3
Defining New Data Types and New Names for Existing Data Types	11-4
Arrays	11-8
Enumerations	11-17
Pointers	11-22
Structures	11-30
Unions	11-39
 <b>Chapter 12. Functions</b>	 <b>12-1</b>
About This Chapter	12-3
main	12-4
Function Definition	12-5
Function Declarations	12-10
Calling Functions and Passing Values	12-12
 <b>Chapter 13. Expressions and Operators</b>	 <b>13-1</b>
About This Chapter	13-3
Grouping and Evaluating Expressions	13-4
Lvalue	13-7
Constant Expression	13-9
Primary Expression	13-11
Unary Expression	13-16
Binary Expression	13-22
Conditional Expression	13-31
Assignment Expression	13-33
Comma Expression	13-35
 <b>Chapter 14. Conversions</b>	 <b>14-1</b>
About This Chapter	14-3
Usual Unary Conversions	14-4
Usual Arithmetic Conversions	14-5
Assignment Conversion	14-7
Explicit Conversions	14-8
 <b>Chapter 15. C Language Statements</b>	 <b>15-1</b>
About This Chapter	15-3
Block	15-4
break	15-6
continue	15-9
do	15-12
Expression	15-14
for	15-15
goto	15-18
if	15-20

Labeled .....	15-23
Null .....	15-24
<b>return</b> .....	15-26
<b>switch</b> .....	15-28
<b>while</b> .....	15-34
 <b>Chapter 16. Preprocessor Statements</b> .....	 <b>16-1</b>
About This Chapter .....	16-3
Preprocessor Statement Format .....	16-4
<b>define</b> .....	16-5
<b>undef</b> .....	16-9
<b>include</b> .....	16-10
Conditional Compilation .....	16-13
line Control .....	16-18
 <b>Appendix A. IBM RT PC C Language</b> .....	 <b>A-1</b>
<b>Appendix B. Portability Considerations</b> .....	<b>B-1</b>
<b>Appendix C. Common Errors in C Programming</b> .....	<b>C-1</b>
<b>Appendix D. Advanced Example Program</b> .....	<b>D-1</b>
<b>Appendix E. Utilities for C Programmers</b> .....	<b>E-1</b>
<b>Appendix F. RT PC Character Codes</b> .....	<b>F-1</b>
<b>Figures</b> .....	<b>X-1</b>
<b>Glossary</b> .....	<b>X-3</b>
<b>Index</b> .....	<b>X-13</b>

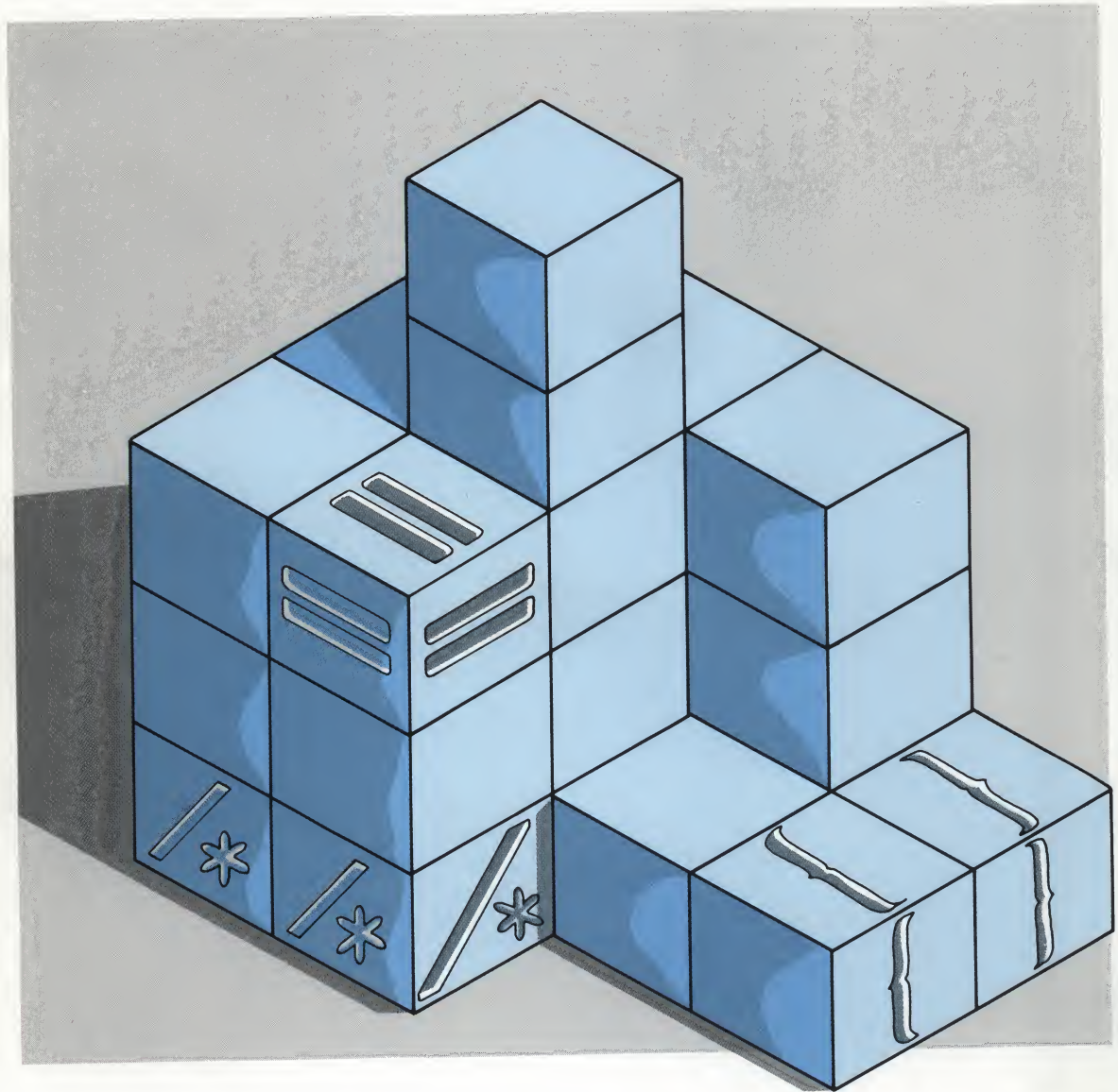
---

## Part 1. Guide



---

## Chapter 1. The Structure and Format of C Language Programs



---

## CONTENTS

About This Chapter .....	1-3
Program Structure .....	1-4
Program Format .....	1-6

---

## About This Chapter

This chapter describes the structure and format of C language programs. You can use this chapter to acquaint yourself with the terms used for each part of a C language program.

---

# Program Structure

---

The C language contains many building blocks that you can use to construct programs. Most of these building blocks fit into one of a few categories. The following paragraphs describe these categories and note examples from the program on page 1-5. (The line numbers that appear to the left of the program are not part of the program code.)

**Comments.** A comment contains text that the compiler ignores. Comments begin with the `/*` characters, end with the `*/` characters, and span any number of lines. Comments cannot be nested. In the example, lines 1 through 3, 9 through 12, 24, 26, and 38 contain comments.

**Preprocessor Statements.** A preprocessor statement contains instructions that the preprocessor interprets before the source file is compiled. Preprocessor statements begin with the `#` symbol in the first column. Directions to the preprocessor follow the `#`. In the example, lines 5 through 7 are preprocessor statements.

**Function Definitions.** A function definition usually contains the code for a single programming task. Functions cannot be nested. A function, however, can call other functions. In the example, lines 14 through 24 define the function `main`, and lines 28 through 38 define the function `is_odd`. Every program contains a function named `main`. When execution begins, the system calls `main`. Thus, `main` controls execution.

A function definition begins with a heading (like line 28) that specifies the type of value the function returns (in this case, `int`), the function name (in this case, `is_odd`), and the parameters (values) the function receives (in this case, `x`). The definition continues with parameter declarations (like line 29) and a block statement (like lines 30 through 38). The block statement contains all the function's data definitions and statements.

**Data Definitions.** A data definition describes a data object, and reserves storage. A data definition can also provide an initial value. Definitions appear outside a function or at the beginning of a block statement. In the example, line 16 defines the variable `number`, and line 31 defines the variable `reply`.

**C Language Statements.** A C language statement contains zero or more expressions. All C language statements, except block statements, end with a `;` symbol. A block statement begins with a `{`, ends with a `}`, and contains any number of statements. In the example, lines 18, 19, 21, 23, 34, 36, 37, 20 through 23, and 33 through 36 are statements.

**Expressions.** An expression represents a value. Expressions can contain constants, variables, operators, and function calls. The example program contains many expressions, such as `reply`, `is_odd(number)` and `(x % 2 != 0)`.

```

1  /* This program accepts an integer as input and determines if
2  ** the integer is odd or even.
3  */
4
5  #include <stdio.h>
6  #define YES 1
7  #define NO 0
8
9  /* The function main accepts an integer as input and passes the
10 ** integer value to the function is_odd. main displays the
11 ** integer and states whether the integer is even or odd.
12 */
13
14 main()
15 {
16     int number, is_odd();
17
18     printf("Enter an integer.\n");
19     scanf("%d", &number);
20     if (is_odd(number) == YES)
21         printf("The integer %d is odd.\n", number);
22     else
23         printf("The integer %d is even.\n", number);
24 } /* End of main */
25
26 /* The function is_odd determines whether an integer is odd. */
27
28 int is_odd(x)
29     int x;
30 {
31     int reply;
32
33     if (x % 2 != 0)
34         reply = YES;
35     else
36         reply = NO;
37     return(reply);
38 } /* End of is_odd */

```

---

## Program Format

---

The C language syntax gives you much freedom in how you format your programs. Although you must distinctly separate constants, identifiers, and keywords, you can separate these and operators by one or more:

- Space characters
- Tab characters
- New-line characters
- Comments.

This formatting flexibility allows you to:

- Continue a statement on the next line
- Place multiple statements on one line
- Use the style of indentation you prefer.

The C language even allows you to begin a new line in the middle of a string constant. In a string constant, however, you must place the `\` symbol at the end of the line to be continued. If space characters appear to the left of a continued string constant, the compiler reads the space characters as part of the string constant. For example:

```
printf("These words surrounded by double quotations are a\  
string constant.");
```

Preprocessor statements are not part of the C language and, thus, follow different formatting rules. The `#` that begins a preprocessor statement must be placed in the first column of a line in your program. (See "Preprocessor Statement Format" on page 16-4.)

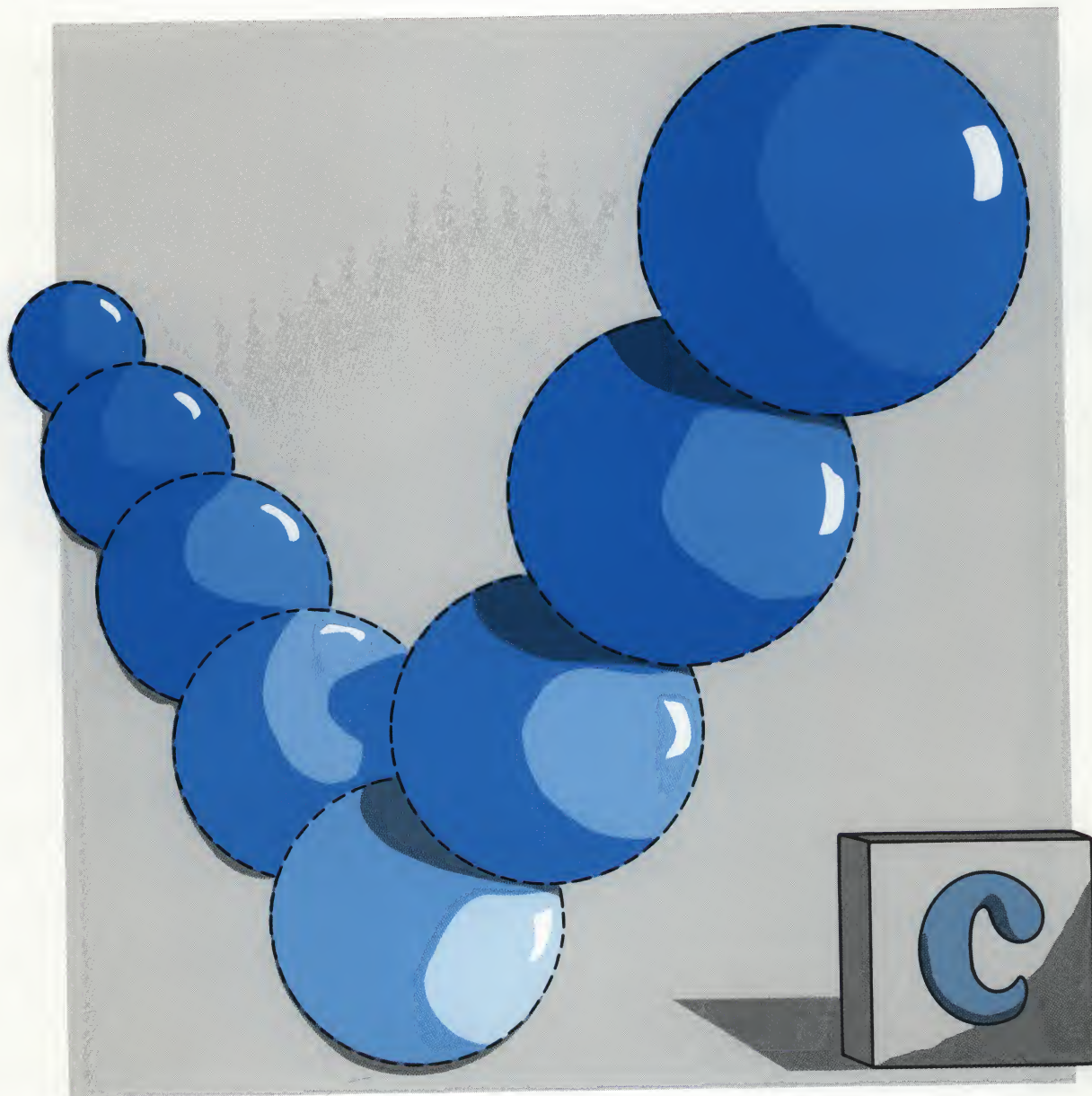
The following example shows the same program formatted two different ways. The C compiler produces the same object code for each format.

```
/* Format Style 1 */  
#include <stdio.h>  
main()  
{  
    printf("This is a C  
program.\n");  
}
```

```
/* Format Style 2 */  
#include <stdio.h>  
main(){  
    printf("This is a C program.\n"); }
```

---

## Chapter 2. Defining Data



---

## CONTENTS

About This Chapter .....	2-3
Defining Constants .....	2-4
Defining Variables .....	2-6
The Storage Class Specifier .....	2-7
The Type Specifier .....	2-8
The Declarator .....	2-9
The Initializer .....	2-10

---

## About This Chapter

The C language allows you to create and use varied types of data. So that the compiler can handle your data properly, your programs must contain a description of each data object. This description is a ***data definition***.

For a constant, the data definition is the mention or usage of the data. For a variable, the data definition is one or more lines of code describing the characteristics of the data.

This chapter explains how to define constants and variables in a C language program.

---

## Defining Constants

---

A **constant** is a data object with a value that is permanently set during compilation. Because a constant never represents anything other than its original value, you can define a constant by using the constant.

The following table briefly describes most C language constants:

Constant	Description	Example
<i>Character</i>	One character enclosed in single quotation marks.	't'
<i>String</i>	Zero or more characters enclosed in double quotation marks. (Although strings are often used like constants, and are here grouped with constants, you can change the value of a string during program execution.)	"No difference."
<i>Integer</i>	A decimal, octal, or hexadecimal constant.	8321292
<i>Decimal</i>	A number containing any digits 0 through 9 that does not begin with 0 (zero).	95437
<i>Octal</i>	The digit 0 (zero) followed by any digits 0 through 7.	0257
<i>Hexadecimal</i>	The characters 0x or 0X (zeroX) followed by any digits 0 through 9 and letters a through f and A through F.	0X9fb374

---

Figure 2-1 (Part 1 of 2). Constants

Constant	Description	Example
<b>Long</b>	An integer constant followed by the letter <b>l</b> (el) or <b>L</b> .	734659L
<b>Floating-Point</b>	A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an <b>e</b> or <b>E</b> , an optional sign (+ or -), and one or more digits 0 through 9. The IBM RT PC C Language compiler allows you to place an optional <b>f</b> or <b>F</b> at the end of a floating-point constant. (Some compilers may generate error messages when they encounter an <b>f</b> or <b>F</b> at the end of a floating-point constant.)	3.5e-4

**Figure 2-1 (Part 2 of 2). Constants**

Chapter 9, "Constants" provides a detailed description of each type of constant.

---

## Defining Variables

---

A **variable** is a data object with a value that can be changed while the program executes. You must define a variable before you can use the variable.

The data definition for a variable names the variable, describes the types of values the variable can represent, reserves storage for the variable, and sometimes provides an initial value for the variable. A typical data definition contains five parts:

**Storage Class Specifier.** A storage class keyword.

**Type Specifier.** A name of a data type.

**Declarator.** An identifier and optional symbols that further describe the data type. (The IBM RT PC C Language compiler allows you to include the optional keyword **volatile** in the declarator portion of a variable definition.)

**Initializer.** The assignment operator followed by an expression (or multiple expressions, for aggregate variables).

**Semicolon.** The ; symbol.

The following data definition describes the variable `dept_goals`. In this definition, `static` is the storage class, `unsigned short` is the type specifier, `dept_goals` is the declarator, and `= 18` is the initializer.

```
static unsigned short dept_goals = 18;
```

You can define multiple variables in one definition by placing a comma after the initializer (or after the declarator, if the variable is not initialized) and specifying another declarator and initializer. If a definition describes multiple variables, each variable has the same storage class and type specifier. For example, the definition:

```
auto int balance=0, payment, credit_limit=500;
```

provides the following description:

Variable	Storage Class	Data Type	Initial Value
balance	auto	int	0
payment	auto	int	undefined
credit_limit	auto	int	500

The following sections describe the parts of a variable data definition.

---

## The Storage Class Specifier

The first part of a data definition is the **storage class specifier**. The storage class describes the location and life span of the values associated with the identifier. The storage class determines the following:

**Storage.** The time the data object is given storage, the time the storage is freed, and the location of storage (register or memory). Variables having the storage class **extern** or **static** maintain storage throughout the execution of a program. Variables having the storage class **auto** or **register** remain active only while the block in which they are defined executes.

**Scope.** The blocks, functions, and files that can access the data object.

**Default Initialization.** The initial value of the data object if an initializer is not specified. Variables having the storage class **extern** or **static** receive 0 (zero) as their default initial value. Variables having the storage class **auto** or **register** receive an undefined default initial value.

The C language supports the following storage classes:

---

Storage Class	Usage of the Storage Class
<b>auto</b>	For a variable that can be accessed by only one block (and its nested blocks) and that maintains storage only while that block executes.
<b>extern</b>	For a variable that can be accessed by several files and that maintains storage throughout the execution of the program.
<b>static</b>	For a variable that can be accessed, depending on where the variable is defined, by only one block (and its nested blocks), one function, or one file. The variable maintains storage throughout the execution of the program.
<b>register</b>	For a heavily used variable that can be accessed by only one block (and its nested blocks) and that remains active only while that block executes.

---

Figure 2-2. Storage Classes

The storage class specifier is optional if the definition contains a type specifier. If you omit the storage class specifier, all variables in the definition receive the default storage class. If the definition is located within a function, the default storage class is **auto**. Otherwise, the default storage class is **extern**.

Chapter 8, "Data Definitions and Storage Classes" describes each storage class.

---

## The Type Specifier

The second part of a data definition is the **type specifier**. The type specifier describes the format of the value found in the identifier's storage. If the variable represents a complex data object, the declarator provides more information about the format of the value.

The following table lists the type specifiers for the primary data types. In the "Type Specifier" column, the optional keywords are followed by the subscript <sub>opt</sub>.

Type Specifier	Description
<b>char</b>	A single character value. When used in an expression, sign extension can occur to a <b>char</b> value.
<b>unsigned char</b>	A single character value on which sign extension cannot occur.
<b>short int</b> <sub>opt</sub>	An integer value that has the same or a smaller number of bits as an <b>int</b> .
<b>unsigned short int</b> <sub>opt</sub>	A nonnegative integer value that has the same number of bits as a <b>short int</b> .
<b>int</b>	An integer value.
<b>unsigned int</b> <sub>opt</sub>	A nonnegative integer value that has the same number of bits as an <b>int</b> .
<b>long int</b> <sub>opt</sub>	An integer value that has the same or a larger number of bits as an <b>int</b> .
<b>unsigned long int</b> <sub>opt</sub>	A nonnegative integer value that has the same number of bits as a <b>long int</b> .
<b>float</b>	A floating-point number value.
<b>double</b>	A floating-point number value that has the same or a larger number of bits as a <b>float</b> .

Type Specifier	Description
<b>long float</b>	Same as <b>double</b> . (The type specifier <b>double</b> is more commonly used than <b>long float</b> .)
<b>long double</b>	A floating-point number value that has the same or a larger number of bits as a <b>double</b> . (Although the IBM RT PC C Language compiler recognizes this type, some compilers may generate error messages when they encounter a <b>long double</b> type specification.)

**Figure 2-3. Type Specifiers**

The type specifier is optional if the definition is located outside a function or if the definition contains a storage class specifier. If you omit the type specifier, all variables in the definition receive the **int** type.

Chapter 10, "Primary Data Types" and Chapter 11, "Complex Data Types" describe the type specifiers that you can use with each data type.

## The Declarator

The third part of a data definition is the **declarator**. You must specify a declarator. A declarator always contains an **identifier** and sometimes contains symbols that further describe the data type.

An identifier is a name that you use to reference a data object. An identifier can contain letters, digits, and underscores. A digit, however, cannot appear as the first character in an identifier. The compiler views uppercase and lowercase letters as different symbols in data definitions that are located inside a function. The following table lists examples of some words that you can use as identifiers and some words that you cannot use as identifiers:

Legal Identifiers	Illegal Identifiers
_account	3-quarter
PROFIT	mark down
profit	TOTAL%
S_gain	phone#
char_3	4256

"Identifiers" on page 7-6 further describes identifiers. For a list of words that you cannot use as identifiers, see "C Language Reserved Words" on page 7-8.

---

If the data type being defined is an array or a pointer, the declarator must contain the special symbols indicated as follows:

Data Type	Description	Example
Array	An optional constant expression within brackets that appears after the identifier.	<code>total[5]</code>
Pointer	An <code>*</code> symbol that appears before the identifier.	<code>*total</code>

See “Declarators” on page 8-8 for further information about declarators. See also “Arrays” on page 11-8 and “Pointers” on page 11-22.

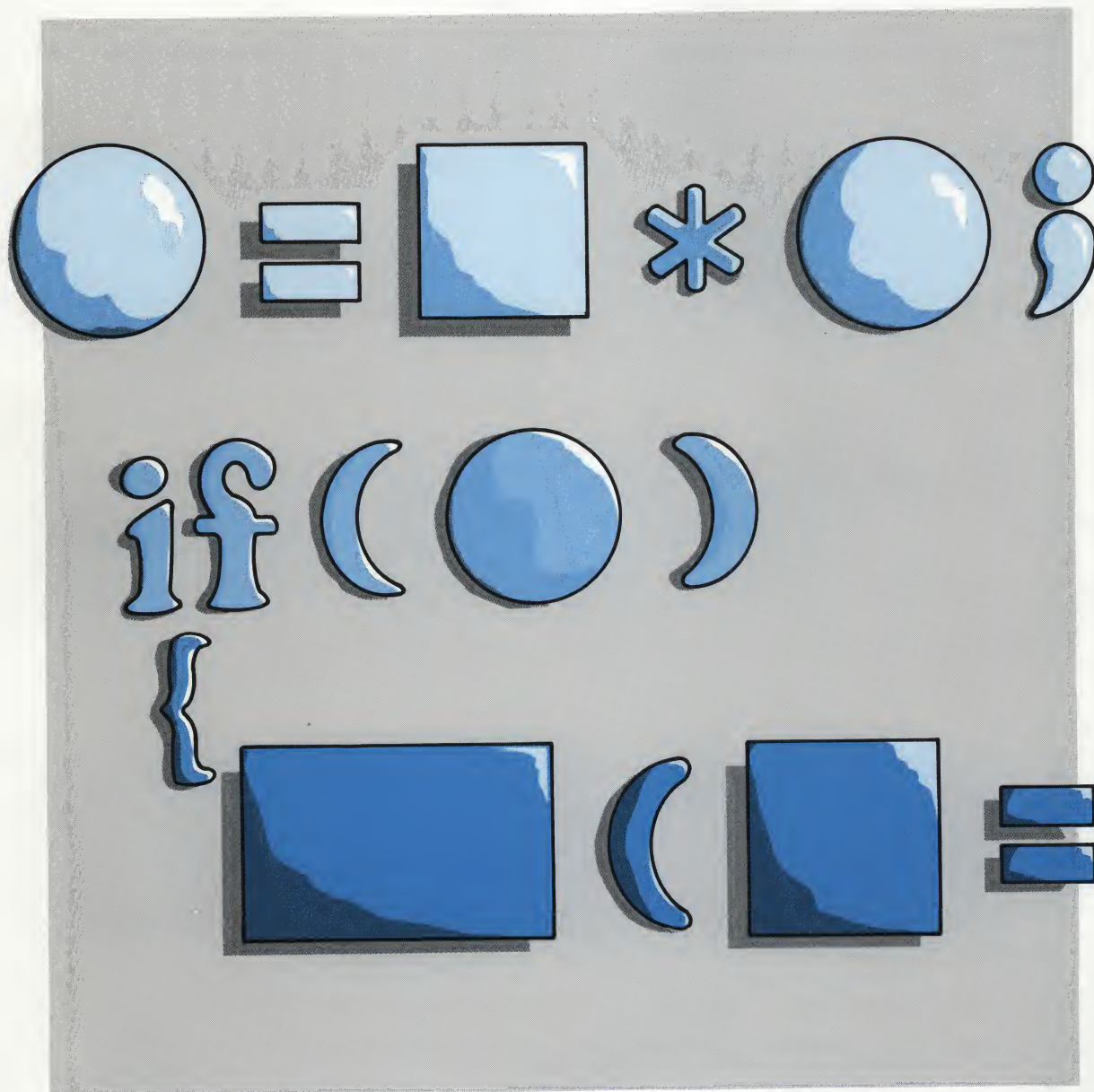
## The Initializer

The fourth part of a data definition is the optional **initializer**. The initializer consists of the assignment operator `=` and an expression. The expression can be surrounded by `{ }` (braces). If the variable being initialized is an **aggregate** (a variable that contains more than one data object, such as an array or a structure), the expression part of the initializer is a brace-enclosed list of expressions (each expression must be separated by a comma).

If you do not specify an initializer, the compiler assigns the value 0 (zero) to **extern** and **static** variables. The C language does not define the initial value of **auto** and **register** variables that are not initialized. You cannot initialize arrays or structures that have the **auto** or **register** storage class or unions.

See “Initializers” on page 8-12 for further information about initializers. Each section that describes a data type in Chapter 10, “Primary Data Types” and Chapter 11, “Complex Data Types” also describes the initializers for that data type.

## Chapter 3. Creating Expressions and Statements



---

## CONTENTS

About This Chapter .....	3-3
Using Operators to Create Expressions .....	3-4
Forming C Language Statements .....	3-6
Block .....	3-6
Expression .....	3-7
Conditional .....	3-8
Looping .....	3-10
Control .....	3-12
Forming C Preprocessor Statements .....	3-14

---

## About This Chapter

This chapter describes how you can put together operators and expressions to form C language statements and preprocessor statements. Each type of operator, expression, and statement is described in “Part 2. Reference.”

---

## Using Operators to Create Expressions

---

**Operators** are symbols (such as +, -, and \*) that represent operations (such as, addition, subtraction, and multiplication). **Expressions** represent values. An expression can contain a single **operand** or several operands and operators. An operand can be an identifier, a constant, or an expression. The following table lists the C language operators:

Operator	Usage	Operator	Usage
[ ]	Array element reference	+	Addition
( )	Expression grouping or function call	-	Subtraction
.	Structure or union member reference	<<	Bitwise shift left
->	Pointer to a structure or union member reference	>>	Bitwise shift right
-	Unary arithmetic negation	<	Less-than
++	Incrementation	>	Greater-than
--	Decrementation	<=	Less-than-or-equal-to
!	Logical negation	>=	Greater-than-or-equal-to
~	Bitwise negation	==	Equality
&	Address reference	!=	Inequality
*	Indirection	&	Bitwise AND
(type)	Type cast	^	Bitwise exclusive OR
sizeof	The size of an object	!	Bitwise inclusive OR
*	Multiplication	&&	Logical AND
/	Division		Logical OR
%	Remainder from division	? :	Conditional evaluation
		=	Assignment
		+=	Addition and assignment
		-=	Subtraction and assignment

Figure 3-1 (Part 1 of 2). Operators

Operator	Usage	Operator	Usage
<code>*</code>	Multiplication and assignment	<code>&amp;</code>	Bitwise AND and assignment
<code>/</code>	Division and assignment	<code>^</code>	Bitwise exclusive OR and assignment
<code>%</code>	Remainder from division and assignment	<code> </code>	Bitwise inclusive OR and assignment
<code>&lt;&lt;</code>	Bitwise shift left and assignment	<code>,</code>	Multiple expression evaluation
<code>&gt;&gt;</code>	Bitwise shift right and assignment		

**Figure 3-1 (Part 2 of 2). Operators**

The following table shows some C language expressions:

Expression	Description
<code>vrbl_x</code>	An identifier.
<code>825</code>	A constant.
<code>vrbl_x = 825</code>	An identifier, an operator, and a constant. (The operator = assigns 825 as the value of <code>vrbl_x</code> .)

Chapter 13, “Expressions and Operators” describes each operator and explains how you can combine operators and operands to form expressions.

---

## Forming C Language Statements

---

**Statements** combine keywords, operators, and operands to describe operations. All statements, except block statements, end with a ; (semicolon).

Each C language statement fits in one of the following categories of statements:

- Block
- Expression
- Conditional
- Looping
- Control.

The following sections describe these categories.

### Block

**Block statements** enable you to group several data definitions, declarations, and statements into a single statement. A block statement begins with a { symbol and ends with a } symbol.

If a block contains data definitions, the definitions must appear at the beginning of the block (before any statements). Any statements, including block statements, occurring within the block can use the variables defined within that block, as well as the variables available to the outer block.

The body of a function must be a block. This block can contain other blocks. The following function contains one block statement. The block begins on line 2 and ends on line 8. Line 3 contains a data definition, and lines 5, 6, and 7 contain statements.

```
1  get_price()
2  {
3      float dollars;
4
5      printf("Enter price: \n");
6      scanf("%f", &dollars);
7      printf("\nPrice = $%.2f\n", dollars);
8  }
                                     /* End block */
```

The following function contains three blocks:

```
1  counter()
2  {                                     /* Begin outer block */
3      int count = 1;
4
5      for (; )
6      {                               /* Begin middle block */
7          if (--count != 0)
8              printf("\ncount = %d\n", count);
9          else
10         {                           /* Begin inner block */
11             printf("\nEnter count: \n");
12             scanf("%d", &count);
13             if (count == 0)
14                 break;
15         }                           /* End inner block */
16     }                               /* End middle block */
17 }                                  /* End outer block */
```

The outer block defines the variable count. Any statement in the outer block can reference count. Because the inner and middle blocks occur within the outer block, the inner and middle blocks can reference count. For more information, see “Block” on page 15-4.

## Expression

**Expression statements** are expressions that end with a ; (semicolon). You can use expression statements to assign the value of an expression to a variable or to call a function. The following lines contain expression statements:

```
result = test1 ! test2;
total = purchase + tax(purchase, region) - discount(purchase, company);
printf("Enter your name: \n");
```

For more information, see “Expression” on page 15-14.

---

## Conditional

**Conditional statements** enable you to specify statements that are to be executed only under certain circumstances. A conditional statement has two parts:

- An expression that serves as a condition
- A statement that serves as an action.

The C language has two types of conditional statements, the **if** or **if...else** statement and the **switch** statement.

The **if** statement is most useful when you want to test for specific conditions. If the condition evaluates to 0 (zero), the action is not carried out and processing continues with the next statement. Otherwise the action is performed. For example:

```
if (level >= 5)      /* condition */
    pay = total * 1.05; /* action   */
```

This **if** statement:

1. Evaluates the condition: `level >= 5`.
2. If the value of the condition is not equal to 0 (zero), performs the action.

For more information, see “**if**” on page 15-20.

When you want to test an expression for several specific constant values, the **switch** statement provides a more concise structure. The **switch** statement enables you to specify a condition and several alternate actions. A **case clause** is an action that has an associated value. Each **case** clause must have a different associated value. A **default clause** is an action that does not have an associated value. You can specify any number of **case** clauses, but only zero or one **default** clause. If the value of the condition matches the value associated with a **case** clause, execution continues with that **case** clause. Otherwise execution continues with the **default** clause. If the statement does not contain a **default** clause, processing continues with the next statement following the **switch** statement. For example:

```

1  switch (choice)
2  {
3      case 'A':
4      case 'a':
5          printf("x + y = %d\n", (x + y));
6          break;
7
8      case 'S':
9      case 's':
10         printf("x - y = %d\n", (x - y));
11         break;
12
13     case 'M':
14     case 'm':
15         printf("x * y = %d\n", (x * y));
16         break;
17
18     default:
19         printf("Enter an 'a', 'A', 's', 'S', 'm', or 'M'.\n");
20         break;
21 }

```

The preceding **switch** statement:

1. Evaluates the condition, choice.
  - If the value of choice is 'A' or 'a', executes lines 5 and 6.
  - If the value of choice is 'S' or 's', executes lines 10 and 11.
  - If the value of choice is 'M' or 'm', executes lines 15 and 16.
  - Otherwise, executes lines 19 and 20.
2. Continues processing with the statement following the **switch** statement.

In a **switch** statement, a **break** statement causes processing to continue with the statement following the **switch** statement. If the previous example did not contain **break** statements, the value 'A' would have caused the **printf** function to be called four times. For more information on the **break** statement, see "Control" on page 3-12 and "break" on page 15-6.

For more information on the **switch** statement, see "switch" on page 15-28.

---

## Looping

*Looping statements* also contain a condition and an action. The action executes as long as the condition evaluates to a nonzero value. The C language contains three types of looping statements:

- **while**
- **do**
- **for**

You can use the **while** statement to evaluate the condition before carrying out the action. If the condition evaluates to 0 (zero), the action is never performed. In the following **while** statement, the action is carried out only while the value of `count` is less than or equal to 3:

```
while (count <= 3)          /* condition          */
{                           /* begin action loop */
    printf("count = %4d\n", count);
    ++count;
}                           /* end action loop  */
```

This **while** statement:

1. Evaluates the condition: `count <= 3`.
2. If the value of the condition is not equal to 0 (zero), performs the action and continues processing with step 1. Otherwise, continues processing with the statement following the **while** statement.

For more information, see "**while**" on page 15-34.

If you want the loop body to execute at least once, use the **do** statement. The **do** statement evaluates the condition after carrying out the action. For example:

```
do                                /* begin do statement */
{                                /* begin action          */
    printf("count = %4d\n", count);
    ++count;
}                                /* end action          */
while (count <= 3);             /* condition        */
```

This **do** statement:

1. Performs the action.
2. Evaluates the condition: `count <= 3`.
3. If the value of the condition is not equal to 0 (zero), continues processing with step 1. Otherwise, continues processing with the statement following the **do** statement.

For more information, see “**do**” on page 15-12.

If you want the action to execute a set number of times, use the **for** statement. The condition part of the **for** statement contains three expressions, each separated by a semicolon. The first expression is evaluated at the beginning of the **for** statement; then the second expression is evaluated. If the value of the second expression equals 0 (zero), processing continues with the statement following the **for** statement. Otherwise, the action is performed, the third expression is evaluated, and the second expression is re-evaluated. Further processing of the **for** statement depends on the value of the second expression in the condition. For example:

```
for (count = 1; count <= 3; ++count) /* initialize, test, increment */
    printf("count = %4d\n", count);   /* action                      */
```

This **for** statement:

1. Initializes count to 1.
2. Evaluates the condition: `count <= 3`.
3. If the value of the condition is equal to 0 (zero), continues processing with the statement following the **for** statement. Otherwise, performs the action, increments the value of `count`, and continues processing with step 2.

You can use a **break** or **continue** statement in the action part of a looping statement. The **break** statement causes processing to continue with the statement following the looping statement. The **continue** statement causes processing to continue with the condition part of the looping statement. For more information, see “**for**” on page 15-15 and “Control” on page 3-12.

---

## Control

**Control statements** change the path of execution. The C language contains four types of control statements:

- **break**
- **continue**
- **return**
- **goto**

The **break** statement stops the processing of the current looping or **switch** statement. Thus, a **break** statement must occur in the action part of a looping statement or in the body of a switch statement. Processing continues with the statement following the looping or **switch** statement. In the following example, the **break** statement on line 4 causes control to pass from the **while** statement to the statement that would follow line 7:

```
1  while (item <= 7)
2  {
3      if ( ( total + price[item] ) > 100)
4          break;
5      total += price[item];
6      item++;
7  }
```

For more information, see “**break**” on page 15-6.

The **continue** statement stops the processing of the action part of the current looping statement. Thus, a **continue** statement must occur in the action part of a looping statement. Processing continues with the re-evaluation of the condition part of the looping statement. In the following example, the **continue** statement on line 4 causes processing to continue with the condition `i <= 20` on line 1:

```
1  for (i = 1; i <= 20; i++)
2  {
3      if ( (i % 2) != 0)
4          continue;
5      printf("even number = %d\n", i);
6  }
```

For more information, see “**continue**” on page 15-9.

---

You can use a **return** statement to stop processing the current function. If an expression follows the word **return**, the value of the expression is passed to the calling function. Execution continues at the statement that called the function. See “**return**” on page 15-26.

If you want to move the path of execution to another part of the current function, you can use the **goto** statement. See “**goto**” on page 15-18.

---

## Forming C Preprocessor Statements

---

*Preprocessor statements* are not part of the C language. They begin with the # symbol and contain specially formatted statements that the preprocessor reads. The **preprocessor** is a program that the **cc** command calls before calling the C compiler. The preprocessor carries out the actions requested by the preprocessor statements. Preprocessor statements direct such actions as macro substitution, conditional compilation, and inclusion of specified files.

The following table lists the major preprocessor keywords and describes the action that each keyword indicates:

---

Preprocessor Keyword	Description
<b>define</b>	Defines a preprocessor macro.
<b>undef</b>	Removes the definition of a macro.
<b>if</b>	Conditionally includes text on the basis of the value of a constant expression.
<b>ifdef</b>	Conditionally includes text if a macro name is defined.
<b>ifndef</b>	Conditionally includes text if a macro name is undefined.
<b>else</b>	Includes some text if the previous preprocessor conditional statement failed.
<b>endif</b>	Ends conditional text.
<b>include</b>	Inserts source text from another file.
<b>line</b>	Alters the current line number. (The compiler uses line numbers in messages to identify where errors occur.)

---

**Figure 3-2. Preprocessor Keywords**

---

In the following example, the **define** preprocessor statement causes the preprocessor to substitute the number 10 for every occurrence of the identifier MAX\_COUNT. The **include** preprocessor statement causes the compiler to insert the library file `stdio.h` into the program source code.

```
#define MAX_COUNT 10  /* define preprocessor statement */

#include <stdio.h>     /* include preprocessor statement */

main()
{
    int count = 0;

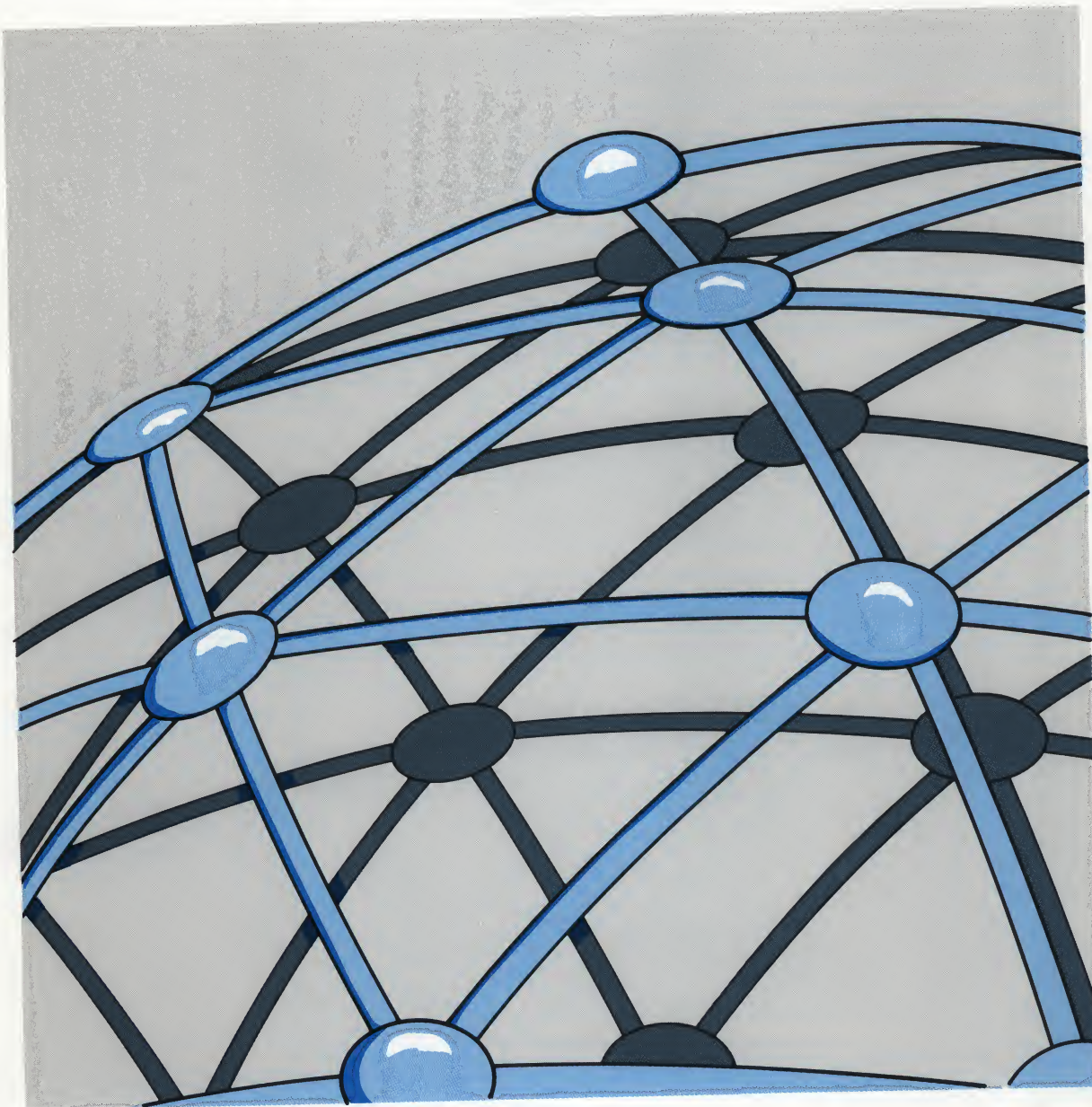
    while (count < MAX_COUNT)
    {
        ++count;
    }
    printf("Maximum count exceeded.\n");
}
```

For more information, see Chapter 16, "Preprocessor Statements."



---

## Chapter 4. Creating and Using Functions



---

## CONTENTS

About This Chapter .....	4-3
Creating Functions .....	4-3
<b>main</b> Functions .....	4-4
Other Functions .....	4-4
Calling Functions and Passing Parameters .....	4-6
Using the C Standard Library Functions .....	4-8
<b>printf</b> .....	4-8
<b>scanf</b> .....	4-14

---

## About This Chapter

This chapter describes how to create functions and how to use the C Language Standard Library of functions. Some standard library input and output functions are described.

---

# Creating Functions

---

Every C language program contains one or more functions. When you run a program, the system calls the function **main**. Thus, all programs must have a function named **main**. The function **main** usually calls other functions.

## main Functions

Every C language program has one function with **main** as its identifier. This *main function* begins and ends program execution.

The following example shows the smallest possible C program:

```
main ()
{
}
```

For more information, see “main” on page 12-4.

## Other Functions

The C language lets you place expressions within expressions and blocks within blocks, but never functions within functions. Although you can reference functions within functions (as a function declaration or a function call), you must place a function definition outside all other functions.

The following program uses three functions: `main`, `print_message`, and `printf`:

```
1  #include <stdio.h>
2
3  main()
4  {
5      void print_message();
6
7      print_message();
8  }
9
10 void print_message()
11 {
12     printf("Function example\n");
13 }
```

---

Lines 3 through 8 contain the definition of `main`, and lines 10 through 13 contain the definition of `print_message`. The `printf` function is defined in the C standard library. The file `stdio.h` (from the C standard library) replaces line 1.

Program execution begins with `main` on line 3. Line 5 declares the function `print_message`. On line 7, `main` calls `print_message`. On line 12, `print_message` calls `printf` and passes the address of the string `"Function example\n"` to `printf`. `printf` produces the following output:

#### Function example

When `printf` finishes executing, control returns to `print_message`. After the `}` on line 13, control returns to `main`. After the `}` on line 8, the example program ends.

If you remove line 7 of the example program, only lines 3 through 6 and 8 execute, and nothing happens.

For more information, see “Function Definition” on page 12-5.

---

## Calling Functions and Passing Parameters

---

A **function call** moves the path of execution from the current function to a specified function. A function call contains the name of the function to which control moves followed by a set of parentheses. The parentheses can hold a list of expressions. Each expression represents a value that the calling function passes to the function specified in the call. These values are called **arguments**.

The following example shows a simple function call. In this example, the calling function does not supply any values to the function `reminder`.

```
reminder()
```

In the following function call, the calling function gives `area` two values: the value of `length` and the value of `width`.

```
area(length, width)
```

When a function receives values, the function names these values and provides declarations for each value. The values that a function receives are called **parameters**. In the following program, `area` is defined as a function that receives two parameters:

```
1  /* program to compute the area of a rectangle */
2  #include <stdio.h>
3
4  main()
5  {
6      int length, width, area();
7
8      printf("Enter a length.\n");
9      scanf("%d", &length);
10     printf("Enter a width. \n");
11     scanf("%d", &width);
12     printf("length = %d\nwidth = %d\n", length, width);
13     printf("area = %d\n", area(length, width));
14 }
```

```
15  int area(side_1, side_2)
16  int side_1, side_2;
17  {
18      int total;
19
20      total = side_1 * side_2;
21      return(total);
22  }
```

Line 15 identifies `area` as a function that receives two values. `area` names these values `side_1` and `side_2`. The variable `side_1` receives the value of `length`. The variable `side_2` receives the value of `width`. Line 16 declares `side_1` and `side_2` as having type `int`.

Line 21 is a **return** statement that sends the calling function (in this case, `main`) the value of `total`. Therefore, the value of this invocation of the function `area` is the value of `total`. Because `total` has the type `int`, the function `area` has the return type `int`.

When run, interaction with the previous program could produce:

```
Output: Enter a length.
Input:  6
Output: Enter a width.
Input:  3
Output: length = 6
        width = 3
        area = 18
```

Parameters are passed by value. Thus, if the values of `side_1` and `side_2` were changed, the values of `length` and `width` would remain unchanged. When you want a called function to access the storage of an argument (rather than the value of the argument), you must pass the address of the argument to the function. The C language treats an array name as a pointer to the first element in the array. Thus, when you pass an array, you pass the address (not the value) of the array.

For more information, see “Calling Functions and Passing Values” on page 12-12 and “**return**” on page 15-26

---

## Using the C Standard Library Functions

---

The C language does not directly support character string operations, input, output, file access, memory allocation, and many other common programming facilities. The C language **standard library** contains functions that provide these facilities. The standard library is packaged with most C language compilers.

Additional libraries are packaged with some C language compilers. When writing portable programs, make sure that all library functions that your programs use are available with the compilers that process your programs.

The following sections describe some standard library functions that support input and output.

*AIX Operating System Technical Reference* describes the RT PC library functions.

### printf

The standard library function **printf** enables you to produce formatted output. The **printf** function accepts two types of parameters:

**Format parameter.** A character string that contains:

- **Plain characters** and **escape sequences**, which are copied to the output stream
- **Conversion specifications**, each of which tells the system how to place the value of zero or more format parameters in the output stream. Each conversion specification begins with a % symbol. The % is followed by conversion modifiers and a conversion code. The **conversion code** specifies the type of the value, as the value is to be printed (in octal format, for example). The **conversion modifiers** specify how the value is to be printed (left justified, for example). If the character following a % is not a conversion modifier or a conversion code, **printf** interprets the % as a plain character to be copied to the output stream.

**Value parameters.** One expression for each conversion specification appearing in the format parameter.

---

A comma separates the format parameter and the list of value parameters. Commas also separate the value parameters. Notice the parts of the following **printf** call:

```
printf("finished goods inventory = %4d\n", fg_inventory);
```

The format parameter is "finished goods inventory = %4d\n". The plain characters are "finished goods inventory = \n". The characters \n are the escape sequence that represents the new-line character. The new-line character causes the next character to be printed on the next line.

The format parameter contains one conversion specification, %4d. The conversion modifier is 4, and the conversion code is d. The 4 specifies that the value of the format parameter is to be right-justified in an area that is four characters wide. The d specifies that the value of the format parameter is to be printed as a decimal value.

The value parameter part of the example is fg\_inventory. If the value of fg\_inventory is 500, the output of the example is:

```
finished goods inventory = 500
```

---

The **printf** function recognizes the following conversion codes:

Conversion Code	Usage of the Conversion Code
<b>d</b>	Displays the value as a decimal number.
<b>u</b>	Displays the value as an unsigned decimal number.
<b>o</b>	Displays the value in octal format.
<b>x</b>	Displays the value in hexadecimal format using the lowercase letters <b>a</b> through <b>f</b> .
<b>X</b>	Displays the value in hexadecimal format using the uppercase letters <b>A</b> through <b>F</b> .
<b>f</b>	Displays the value in floating-point format. If the precision is not specified, six digits to the right of the decimal are displayed. This format can be used for <b>float</b> and <b>double</b> objects.
<b>e</b>	Displays the value in exponential notation format. The displayed value contains the sign - (if appropriate) followed by one digit, a decimal point, several more digits (six digits, unless a conversion modifier specifies a precision), the character <b>e</b> (lowercase), a sign (+ or -), and an exponent.
<b>E</b>	Displays the value in exponential notation format. The displayed value contains the sign - (if appropriate) followed by one digit, a decimal point, several more digits (six digits, unless a conversion modifier specifies a precision), the character <b>E</b> (uppercase), a sign (+ or -), and an exponent.
<b>g</b>	Displays the <b>float</b> or <b>double</b> object in the shortest form, as indicated by the <b>f</b> or <b>e</b> conversion characters.
<b>G</b>	Displays the <b>float</b> or <b>double</b> object in the shortest form, as indicated by the <b>f</b> or <b>E</b> conversion characters.
<b>c</b>	Displays the value as a single character.
<b>s</b>	Displays the value as a character string. If the precision is not specified, the character string ends at the first null character.

**Figure 4-1. printf Conversion Codes**

The following table describes the **printf** conversion modifiers:

Conversion Modifier	Usage of the Modifier
-	Left-justifies the value in the output area.
+	Precedes the value with a +, if the value is positive. (All negative values are automatically preceded by a -.)
<i>blank</i>	Places a space character before positive numbers.
#	For the <b>e</b> , <b>E</b> , <b>f</b> , <b>g</b> , and <b>G</b> codes, the # modifier causes a decimal point to be placed in the output, even if no digits follow the decimal point. For the <b>g</b> and <b>G</b> codes, the # causes trailing zeroes to be placed in the output. For the <b>o</b> code, the # modifier increases the precision to force the first digit of the result to be zero. For the <b>x</b> and <b>X</b> codes, the # modifier causes nonzero results to contain the prefix <b>0x</b> (for the <b>x</b> code) or <b>0X</b> (for the <b>X</b> code).
<i>field-width</i> or *	Specifies the minimum width of the output area. If the output value does not fill the output area, the output value is padded. Padding occurs on the left side of the output value, unless the conversion code is preceded by the - modifier. If the * symbol appears in the place of the field-width, a value parameter that specifies the field-width must precede the value parameter that provides the output value.
<i>. precision</i> or .*	For the <b>d</b> , <b>o</b> , <b>u</b> , <b>x</b> , and <b>X</b> codes, the precision modifier specifies the minimum number of digits to appear in the output value. For the <b>e</b> , <b>E</b> , and <b>f</b> codes, the precision modifier specifies the number of digits to appear after the decimal point. For the <b>g</b> code, the precision modifier specifies the maximum number of significant digits to appear in the output value. For the <b>s</b> code, the precision modifier specifies the maximum number of characters to appear in the output value. If the * symbol appears in the place of the precision, a value parameter that specifies the precision must precede the value parameter that provides the output value. If the conversion code immediately follows the . modifier, the precision is 0 (zero).
<b>l</b> (el)	Treats the value as a <b>long int</b> .

Figure 4-2. **printf** Conversion Modifiers

The following program calls **printf** several times to show different ways of formatting **int**, **float**, and **double** variables. Exclamation marks separate the output areas in the **printf** function calls. The exclamation marks help to show field-width in the program output.

```
/* Various printf formats for integer and floating-point numbers */
main()
{
    int w = 1666;
    float x = 123.456789;
    float y = -546.321;
    double z = 123.456789;

    printf("line a INTEGERS, FLOATS, AND DOUBLES:\n\n");
    printf("line b !%d!%f!%f!%f!\n", w, x, y, z);
    printf("line c !%5d!%11f!%9f!%11f!\n", w, x, y, z);
    printf("line d !%*d!%*f!%*f!%*f!\n", 5, w, 11, x, 9, y, 11, z);
    printf("line e !%-5d!%-11f!%-9f!%-11f!\n", w, x, y, z);
    printf("line f !%3d!%10f!%8f!%10f!\n", w, x, y, z);
    printf("line g !      !%.2f!%.2f!%.2f!\n", x, y, z);
    printf("line h !      !%7.2f!%7.2f!%7.2f!\n", x, y, z);
    printf("line i !      !%7.0f!%7.0f!%7.0f!\n", x, y, z);
    printf("line j !      !%7.10f!%7.10f!\n", x, z);
    printf("line k !      !%*.*f!%*.*f!\n", 7, 10, x, 7, 10, z);
}
```

The preceding example produces the following output:

```
line a INTEGERS, FLOATS, AND DOUBLES:

line b !1666!123.456787!-546.320984!123.456789!
line c ! 1666! 123.456787!-546.320984! 123.456789!
line d ! 1666! 123.456787!-546.320984! 123.456789!
line e !1666 !123.456787 !-546.320984!123.456789 !
line f !1666!123.456787!-546.320984!123.456789!
line g !      !123.46!-546.32!123.46!
line h !      ! 123.46!-546.32! 123.46!
line i !      !    123!   -546!    123!
line j !      !123.4567871094!123.4567890000!
```

---

The following program calls **printf** several times to show different ways of formatting characters and strings.

```
/* Various printf formats for characters and strings */

main()
{
    int letter = 'K';
    static char strng[ ] = "Good Day";

    printf("line a CHARACTERS AND STRINGS:\n\n");
    printf("line b !c!s!\n", letter, strng);
    printf("line c !2c!9s!\n", letter, strng);
    printf("line d !*c!*s!\n", 2, letter, 9, strng);
    printf("line e !-2c!-9s!\n", letter, strng);
    printf("line f !0c!5s!\n", letter, strng);
    printf("line g !.5s!%10.5s!%-10.5s!\n", strng, strng, strng);
    printf("line h !%.s!%*.s!%-*.s!\n", 5, strng, 10,5, strng,
        10,5, strng);
}
```

This example produces the following output:

```
line a CHARACTERS AND STRINGS:

line b !K!Good Day!
line c ! K! Good Day!
line d ! K! Good Day!
line e !K !Good Day !
line f !K!Good Day!
line g !Good !      Good !Good      !
line h !Good !      Good !Good      !
```

---

## scanf

The standard library function **scanf** enables your programs to accept formatted input. The **scanf** function accepts two types of parameters:

**Format parameter.** A character string that contains conversion specifications. Each conversion specification tells the system how to interpret the next item in the input stream. Each conversion specification contains a % symbol and a conversion code. You can place some optional conversion modifiers between the % and the conversion code. The conversion code specifies how to interpret the input (as an octal number, for example). The conversion modifiers further specify how to interpret the input (as a **short int**, for example). Each conversion specification can be separated by any number of space characters or tab characters. The entire format parameter must be enclosed in double quotation marks.

**Pointer parameters.** One expression for each conversion code in the format parameter that is not modified by the \* symbol. Each expression must be a pointer to a variable. The expression can be an identifier (if the identifier is a pointer, such as an array name) or an & followed by an identifier.

A comma separates the format parameter and the list of pointer parameters. Commas also separate the pointer parameters. Notice the parts of the following **scanf** call:

```
scanf("%ld", value);
```

In this example, the format parameter is "%ld" and the pointer parameter is `value`. The specification %ld causes **scanf** to treat `value` as a **long** decimal number.

---

The **scanf** function recognizes the following conversion codes:

Conversion Code	What scanf Expects From Input
<b>d</b>	A decimal number.
<b>u</b>	An unsigned decimal number.
<b>o</b>	An octal number.
<b>x</b>	A hexadecimal number.
<b>e, f, g</b>	A floating-point number.
<b>c</b>	A single character.
<b>s</b>	A character string. The string ends with the next white space character. The pointer parameter should be a character pointer that points to an array of characters large enough to accept the string and a null character. The null character is added automatically.
<b>[<i>characters</i>]</b>	A character string. The characters accessed by the corresponding pointer parameter can contain only the characters listed in the brackets. The value of the pointer parameter is delimited by the first occurrence of a character not listed in the brackets.
<b>[<i>^characters</i>]</b>	A character string. The characters accessed by the corresponding pointer parameter can contain only the characters not listed in the brackets. The value of the pointer parameter is delimited by the first occurrence of a character listed in the brackets.

**Figure 4-3.** scanf Conversion Codes

---

The following table describes the **scanf** conversion modifiers:

Conversion Modifier	Usage of the Modifier
<i>field-width</i>	Specifies the maximum number of characters in the input area.
<b>*</b>	Skips over the input area. The values that appear in this input area are not assigned storage. Therefore, no pointer parameter should be specified for the conversion code that the <b>*</b> modifies.
<b>h</b>	Treats the value as a pointer to a <b>short int</b> . <b>h</b> can modify the following codes only: <b>d, u, o, x</b> .
<b>l (el)</b>	Treats the value as a pointer to a <b>long int</b> or to a <b>double</b> . <b>l</b> can modify the following codes only: <b>d, e, f, g, u, o, x</b> .

**Figure 4-4. scanf Conversion Modifiers**

For each conversion code specified in a **scanf** call, **scanf** reads an input stream. If a field-width is provided, **scanf** reads from standard input until a white space character occurs or until the size of the input area matches the field-width, whichever comes first. If no field-width is provided, **scanf** reads from standard input until a white space character occurs. White space characters are: space characters, tab characters, and new-line characters.

---

The following program calls **scanf**, but does not provide field-widths for any of the conversion codes:

```
#include <stdio.h>

main()
{
    int month;
    int day;
    int year;

    printf("Enter month, day, and year:\n");
    scanf("%d%d%d", &month, &day, &year);
    printf("month = %d\n", month);
    printf("day = %d\n", day);
    printf("year = %d\n", year);
}
```

Interaction with this program could produce the following session:

```
Output:  Enter month, day, and year:
Input:   12 19 1985
Output:  month = 12
         day = 19
         year = 1985
```

In this session the system user enters the input in this format:

```
12 19 1985
```

Each input area is separated by a space character. Because a new-line character is a white space character, the user could have entered:

```
12
19
1955
```

---

The conversion code **c** provides an exception to the way **scanf** treats white space. **c** causes **scanf** to read the next character as the input, regardless of whether the next character is a white space character. If you want **scanf** to read the next non-white space character, you must place a space character before the **%c** specification. The following example contains a **%c** specification that is not preceded by a space character:

```
#include <stdio.h>

main()
{
    float number;
    char letter;

    printf("Enter number and letter:  \n");
    scanf("%f%c", &number, &letter);
    printf("number = %f\n", number);
    printf("letter = %c\n", letter);
}
```

Interaction with this program could produce the following session:

```
Output:  Enter number and letter:
Input:   12.5 k
Output:  number = 12.500000
         letter =
```

In this example, **letter** receives the value of a space character. If the input was changed to **12.5k** or the format parameter was changed to **"%d %c"**, **letter** would received the value of **k**.

---

In the following example, the `scanf` call contains an assignment suppression modifier and a field-width modifier:

```
#include <stdio.h>

main()
{
    char first_name[20];
    char last_name[20];
    int age;

    scanf("%s %*c %s %2d", first_name, last_name, &age);
    printf("first_name = %s\n", first_name);
    printf("last_name = %s\n", last_name);
    printf("age = %d\n", age);
}
```

Interaction with this program could produce the following session:

```
Input:  Cheryl A Chambers 2955
Output: first_name = Cheryl
        last_name = Chambers
        age = 29
```

This invocation of the program assigns the string Cheryl to `first_name`, skips over the character A, assigns the string Chambers to `last_name`, and assigns the decimal number 29 to `age`.

Array names are pointers to the first element of the array. Therefore, the array names `first_name` and `last_name` are not preceded by the `&` operator in the pointer parameter.



---

## Chapter 5. Compiling, Linking, and Running a Program



---

## CONTENTS

About This Chapter .....	5-3
Compiling and Linking .....	5-4
Running .....	5-7

---

## About This Chapter

This chapter presents an overview of compiling, linking, and running C language programs. This chapter does not replace the description of the `cc` (C compiler) command in the *AIX Operating System Commands Reference*. See the *AIX Operating System Commands Reference* for complete details about the compiler command.

---

## Compiling and Linking

---

The **cc** command translates C language programs into code that the system can run. The **cc** command can process several types of files. Two types are:

**Source files.** Files that were not translated by the compiler or the assembler. The name of a source file must end with a **.c** suffix. For example:

`payroll.c`

**Object files.** Files that were translated by the compiler or the assembler. The name of an object file must end with a **.o** suffix. For example:

`utility.o`

A simple form of the **cc** command is:

`cc account.c`

This command sends the contents of the source file `account.c` through:

1. The preprocessor
2. The compiler
3. The assembler
4. The link editor

and produces the executable file **a.out**.

When you compile more than one file, **cc** creates an object file for each source file besides the executable file. For example, the command:

`cc payroll.c tax.c`

produces the following files:

`payroll.o`

`tax.o`

`a.out`

The `a.out` file is the executable file. Both `payroll.o` and `tax.o` are object files.

You can use the **cc** command to link object files with other files. For example, you can recompile `tax.c` and link it with `payroll.o` with the following command:

`cc tax.c payroll.o`

The **cc** command sends the contents of `tax.c` through the preprocessor, the compiler, the assembler, and the link editor. The **cc** command sends the contents of `payroll.o` through the link editor only.

---

You can change the way the **cc** command processes files by specifying **compiler flags** in the command. Except for the **-l** (el) flag, you can place compiler flags anywhere in the **cc** command. The **-l** flags must be specified as the last entries on the command line. The following table describes some of the compiler flags. See *AIX Operating System Commands Reference* for a complete list of compiler flags.

Flag	Effect of the Flag
<b>-c</b>	The <b>cc</b> command does not send the object file to the link editor. The <b>cc</b> command creates an object file with a <b>.o</b> suffix. An object file, however, is not executable unless it has been processed by the link editor. The following command creates the object file <b>prime.o</b> :  <code>cc -c prime.c</code>
<b>-Dname</b> <b>-Dname=def</b>	The preprocessor defines <i>name</i> as a <b>define</b> directive. If <b>=def</b> is omitted, the preprocessor defines <i>name</i> as <b>1</b> (one). Otherwise, the preprocessor defines <i>name</i> as <i>def</i> . You can place space characters in <i>def</i> by surrounding <i>def</i> with double quotation marks. See “ <b>define</b> ” on page 16-5.  The following command defines <i>order</i> as <b>1</b> and compiles <b>sort.c</b> :  <code>cc -Dorder sort.c</code>  The following command defines <i>order</i> as <b>4</b> and compiles <b>sort.c</b> :  <code>cc -Dorder=4 sort.c</code>  The following command defines <i>order</i> as <b>a b c</b> and compiles <b>sort.c</b> :  <code>cc -Dorder="a b c" sort.c</code>
<b>-Idir</b>	The preprocessor looks first in <i>dir</i> , then in the standard directories for <b>include</b> files with names that do not begin with the <b>/</b> symbol. If you specify several <b>-Idir</b> flags, the preprocessor searches the directories in the order in which you list them on the command line. See “ <b>include</b> ” on page 16-10.
<b>-l</b> <b>-lkey</b>	The link editor searches the specified library file, for the file <b>libkey.a</b> . If you do not specify <i>key</i> , the link editor selects <b>libc.a</b> , the standard system library for C programs. The link editor searches for this file in the directory specified by the <b>-L</b> flag, then in <b>/lib</b> and <b>/usr/lib</b> . The link editor searches library files in the order in which you list them on the command line. If you specify this flag, you must make it the last entry on the command line.

---

Figure 5-1 (Part 1 of 2). **cc** Flags

---

Flag	Effect of the Flag
<b>-Ldir</b>	The link editor looks in <i>dir</i> for files specified by <b>-l</b> keys. If the file is not located in <i>dir</i> , the link editor searches the standard directories.
<b>-ofilename</b>	The link editor places the executable code in the file specified by <i>filename</i> . If you do not use the <b>-o</b> flag, the link editor stores the executable code in the file <b>a.out</b> . The following command compiles the file <code>account.c</code> and stores the executable code in the file <code>account.out</code> :  <code>cc -oaccount.out account.c</code>
<b>-O</b>	The <code>cc</code> command sends the output of the compiler through the optimizer. The optimized code then goes through the assembler.

---

**Figure 5-1 (Part 2 of 2). cc Flags**

---

## Running

---

You can run a program by typing the name of the executable file and pressing the new-line (**Enter**) key. If the name of the executable file is **a.out**, type:

**a.out**

and press the new-line (**Enter**) key.

If you use the **-ofilename** flag in the **cc** command to specify a name for the executable file, you can run the executable file by entering the *filename*. For example, the following command creates the executable file **size**:

**cc -o size sizing.c**

You can run **size** by entering:

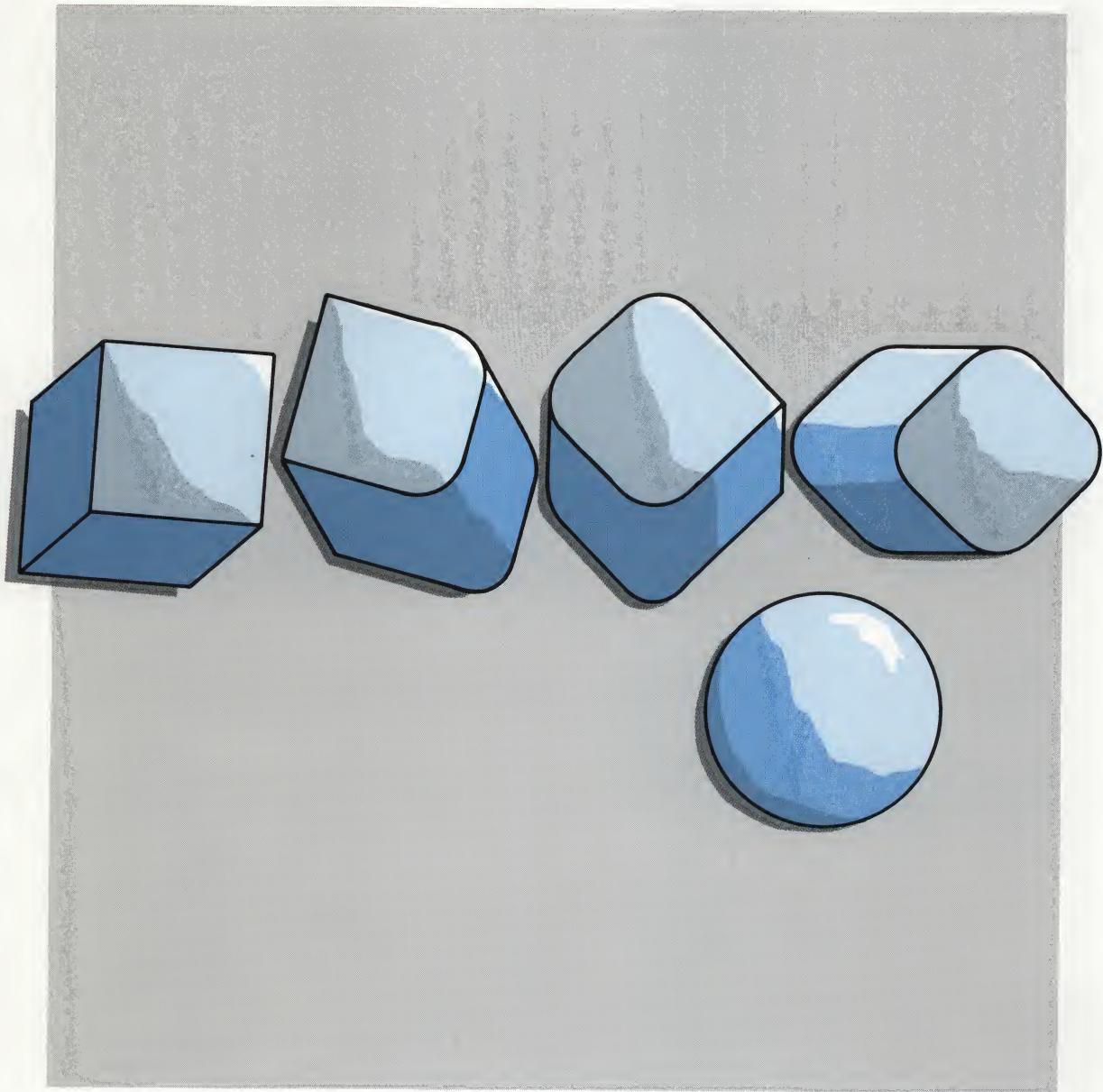
**size**

If your program meets a problem, such as an infinite loop, you may need to stop the program while it is running. Some systems provide an interrupt key (or an interrupt key sequence), which halts the execution of the current program. *Using the AIX Operating System* describes the interrupt key sequence for the AIX Operating System.



---

## Chapter 6. Debugging a Program



---

## CONTENTS

About This Chapter .....	6-3
Understanding Error Messages .....	6-4
Using System Utilities to Detect Errors .....	6-5
<b>lint</b> .....	6-5
<b>cb</b> .....	6-8

---

## About This Chapter

This chapter tells you which system programs produce error messages and where these messages are explained. This chapter also describes two utility programs (**lint** and **cb**) that can help you find errors. *AIX Operating System Commands Reference* provides more complete descriptions of these utility programs.

---

## Understanding Error Messages

---

When you compile a program, you might receive ***error messages***. Error messages warn you that the command you entered or the code you wanted compiled contains syntax errors. When you receive error messages, an executable file may not have been created.

The preprocessor, compiler, assembler, linker, and operating system report errors during compilation. *Messages Reference* explains the preprocessor, compiler, assembler, and linker messages, as well as other operating system messages.

---

## Using System Utilities to Detect Errors

---

The AIX Operating System contains several programs that can help you find and fix errors in C language programs. Two of these programs are discussed in this section, the **lint** code checker and the **cb** reformatter.

### **lint**

The **lint** program checks source files for potential coding problems. Some of these problems are:

- Syntax errors
- Variables that are defined but never used
- Variables that are used before they are initialized
- Function return values that are not used
- Function parameters that are mismatched
- Data types that are used incorrectly
- Unclear operator precedence
- Parts of a program that cannot be reached
- Nonportable code.

If **lint** finds any problems in a program, it prints error messages. You can run the **lint** program by entering **lint** followed by the names of the files you want checked. For example:

```
lint file1 file2
```

You can change the way **lint** checks files by specifying flags between **lint** and the list of files to be checked. Some of the flags that **lint** accepts are:

Flag	Effect of the Flag
-a	Suppresses messages about assignments of long values to variables that are not long.
-b	Suppresses messages about unreachable <b>break</b> statements.
-Dname -Dname=def	Defines <i>name</i> as a <b>#define</b> directive. If <i>=def</i> is omitted, <i>name</i> is defined as 1 (one). Otherwise, <i>name</i> is defined as <i>def</i> . You can place space characters in <i>def</i> by surrounding <i>def</i> with double quotation marks. See “ <b>define</b> ” on page 16-5.
-p	Checks for portability to other C language dialects.
-u	Suppresses messages about functions and external variables that are either used and not defined or defined and not used. You can use this flag to run <b>lint</b> on a subset of files of a larger program.
-v	Suppresses messages about function parameters that are not used.
-x	Suppresses messages about variables that have external definitions but are never used.

**Figure 6-1. lint Flags**

When **lint** is run on the following program, **lint** generates some warnings. Only the warning about line 14 being unreachable would be generated by both **lint** and **cc**.

```
1  main()
2  {
3      int x, y, z;
4
5      x = y;
6      z = 1;
7      func(z);
8  }
9
```

```

10     func(p)
11     int p;
12     {
13         return(p + p);
14         p = 1;
15     }

```

If the preceding program were stored in the file `try_lint.c`, the following session could occur with **lint**:

```

Input:  lint try_lint.c
Output: try_lint.c
=====
(5)  warning: y may be used before set
(5)  warning: x set but not used in function main
(15) warning: function func has return(e); and return;
warning: statement not reached
      (14)

=====
function returns value which is always ignored
      func

```

The first warning says that `y` is not assigned a value before the value of `y` is referenced on line 5. Because `y` has the storage class **auto** and is not initialized, the default value of `y` is undefined.

The second warning says that `x` is assigned a value on line 5, but the value of `x` is never used within `main`. Because `x` has the storage class **auto**, `main` is the only function that can directly access the value of `x`.

The third warning says that the function `func` has two lines that return control to `main`. Line 13 is a `return` statement that is always reached. Line 14 is an expression statement that is never reached. Since the expression statement comes between the `return` statement and the `}` symbol that ends the function, line 15 serves as a second `return` statement.

The last warning says that the function `func` returns a value that `main` does not use.

For more information on **lint**, see *AIX Operating System Commands Reference* or *AIX Operating System Programming Tools and Interfaces*.

---

## cb

The **cb** program re-formats source code into a consistent format. Having easily readable source code is important when trying to find syntax and logic errors and when trying to update code. If you maintain or create source code that is not in an easily readable format, running **cb** on the source files re-formats the code within the files. You can run **cb** by entering **cb** followed by the names of the files you want reformatted. For example:

```
cb file1 file2
```

The system writes the re-formatted source code to standard output (usually the screen). If you want the system to store the re-formatted source code in a file, you can use the following command:

```
cb file1 file2 > file3
```

This command reads the code in `file1` and `file2`, reformats the code, and stores the reformatted code in `file3`. This command does not change the contents of `file1` and `file2`.

You can change the way **cb** formats files by specifying flags between **cb** and the list of files to be formatted. Some of the flags that **cb** accepts are:

Flag	Effect of the Flag
-s	Formats the source code according to a well-known style (not the style used in examples throughout this book).
-j	Joins lines that are split.
-l <i>length</i>	Splits lines that are longer than <i>length</i> . (You must place a space character between <i>-l</i> and <i>length</i> .)

Figure 6-2. **cb** Flags

---

The following program has an inconsistent format. Because this program is small and does not contain many nested levels of code, this program may not be very hard to read. A larger and more complicated program that is inconsistently formatted can be difficult to read.

```
#include <stdio.h>
main()
{
printf("This is a C program.\n");
}
```

If the preceding program were stored in the file `try_cb.c`, you could re-format the program by entering:

```
cb try_cb.c
```

This command would cause the system to write the following in standard output:

```
#include <stdio.h>
main()
{
    printf("This is a C program.\n");
}
```

For more information on **cb**, see *AIX Operating System Commands Reference* or *AIX Operating System Programming Tools and Interfaces*.



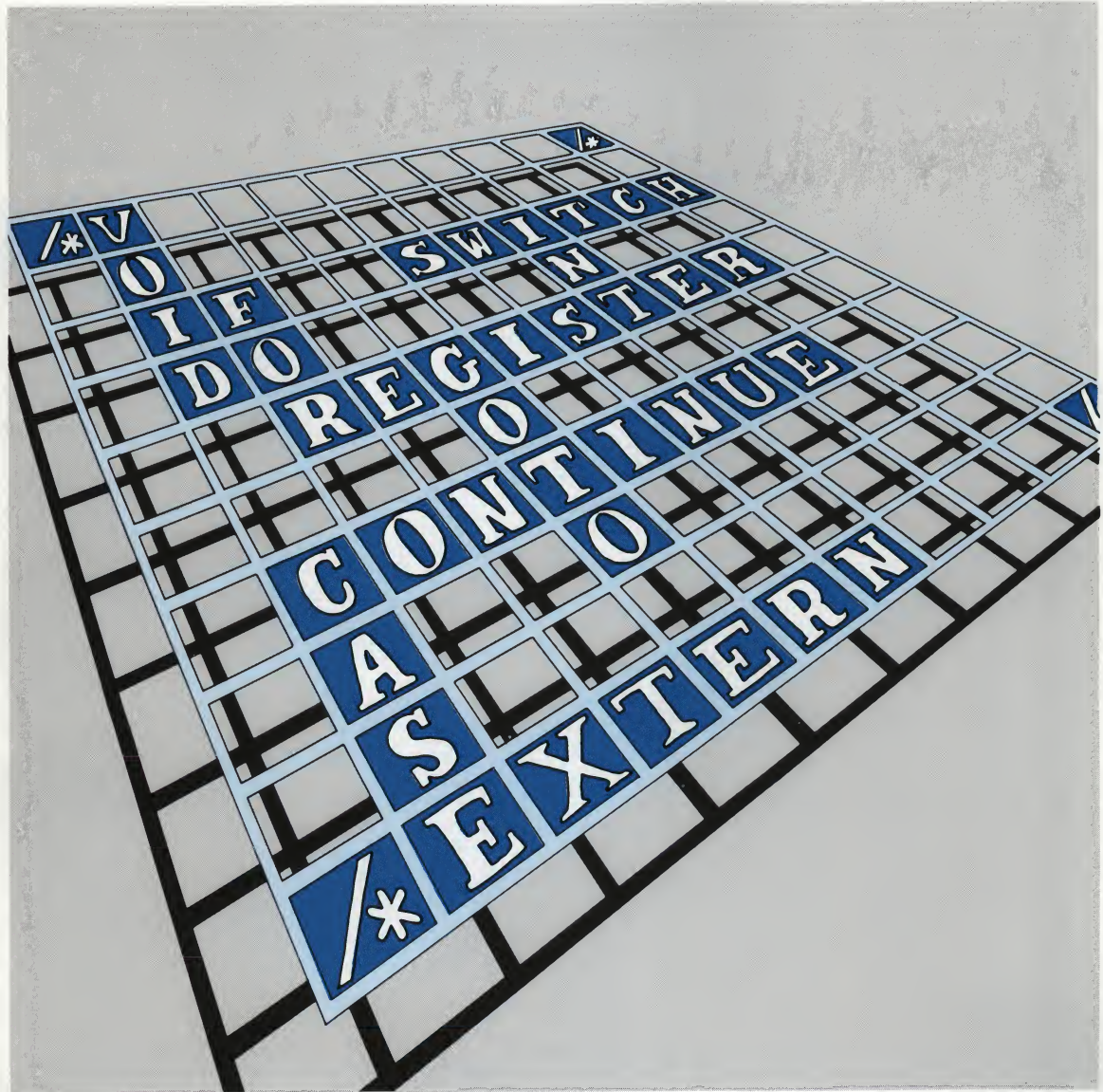
---

## Part 2. Reference



---

## Chapter 7. Comments, Identifiers, and Reserved Words



---

## CONTENTS

About This Chapter .....	7-3
Comments .....	7-4
Identifiers .....	7-6
C Language Reserved Words .....	7-8

---

## About This Chapter

This chapter describes comments, identifiers, and reserved words.

---

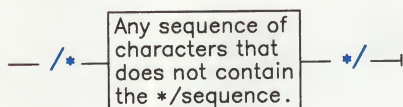
## Comments

---

You can use **comments** to explain code to yourself and to other programmers who maintain the code. Comments are notes that the compiler ignores.

Comments begin with the `/*` characters, end with the `*/` characters, and span any number of lines. You can place comments anywhere that the C language allows blanks, tabs, and new-line characters. Comments have the form:

*comment*



In the following program, line 6 serves as a comment:

```
1  #include <stdio.h>
2
3  main()
4  {
5      printf("This program has a comment.\n");
6      /* printf("This program has a comment.\n"); */
7  }
```

Because the compiler ignores line 6, the output of this program is:

This program has a comment.

You cannot nest comments. Each comment ends at the first occurrence of `*/`.

---

In the following example, everything that the compiler recognizes as a comment is shaded:

```
1  /* A program with nested comments. */
2
3  #include <stdio.h>
4
5  main()
6  {
7      test_function();
8  }
9
10 test_function()
11 {
12     int number;
13     char letter;
14     /*
15      number = 55;
16      letter = 'A';
17      /* number = 44; */
18     */
19 }
```

In `test_function` the programmer tries to remove lines 14 through 18 by making these lines a comment. The compiler reads the `/*` in line 14 through the `*/` in line 17 as a comment and line 18 as C language code. Because line 18 is not a legal C language statement, the compiler reports an error at line 18.

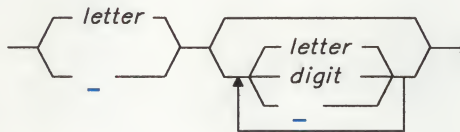
---

# Identifiers

---

Identifiers provide names for functions, data objects, and lines of code. An identifier has the form:

*identifier*



The C language does not limit the number of characters in an identifier. However, most C language compilers consider only the first several characters of long identifiers to be significant. If you make the first eight characters unique for internal data objects and the first six characters unique for external data objects, most compilers will accept your identifiers. The IBM RT PC C Language compiler considers at least the first 64 characters in internal and external identifiers as significant. Some programs that the compiler command calls and some of the functions that your programs call may further limit the number of significant characters in identifiers.

For identifiers that represent internal data objects, C compilers view uppercase and lowercase letters as different symbols. Thus, PROFIT and profit represent different internal data objects.

For identifiers that represent external data objects, some compilers view uppercase and lowercase letters as equivalent symbols. Thus, CHARGE and charge can represent the same external data object or different external data objects. The IBM RT PC C Language compiler views uppercase and lowercase letters as different symbols. Thus, CHARGE and charge represent different external data objects.

Avoid creating identifiers that begin with an - (underscore) for function names and variable names.

---

The following table lists some C language identifiers:

<b>Legal Identifier</b>	<b>What a Compiler That Considers the First Eight Characters Views</b>
product1	product1
product2	product2
product_1	product_
product_2	product_

---

## C Language Reserved Words

---

The C language reserves some words for special usage. You cannot use these words as identifiers. Only the lowercase versions of the words are reserved. The following figure lists the C language reserved words:

---

<b>auto</b>	<b>else</b>	<b>long</b>	<b>typedef</b>
<b>break</b>	<b>enum</b>	<b>register</b>	<b>union</b>
<b>case</b>	<b>extern</b>	<b>return</b>	<b>unsigned</b>
<b>char</b>	<b>float</b>	<b>short</b>	<b>void</b>
<b>continue</b>	<b>for</b>	<b>sizeof</b>	<b>while</b>
<b>default</b>	<b>goto</b>	<b>static</b>	
<b>do</b>	<b>if</b>	<b>struct</b>	
<b>double</b>	<b>int</b>	<b>switch</b>	

---

Figure 7-1. The C Language Reserved Words

Some C language compilers reserve additional words. If you are writing portable programs, avoid using words beginning with the `_` (underscore) character for external identifiers. Also avoid using the following words as identifiers:

---

<b>asm</b>	<b>entry</b>	<b>inline</b>	<b>signed</b>
<b>class</b>	<b>far</b>	<b>near</b>	<b>volatile</b>
<b>const</b>	<b>fortran</b>	<b>pragma</b>	
<b>defined</b>	<b>generic</b>	<b>public</b>	

---

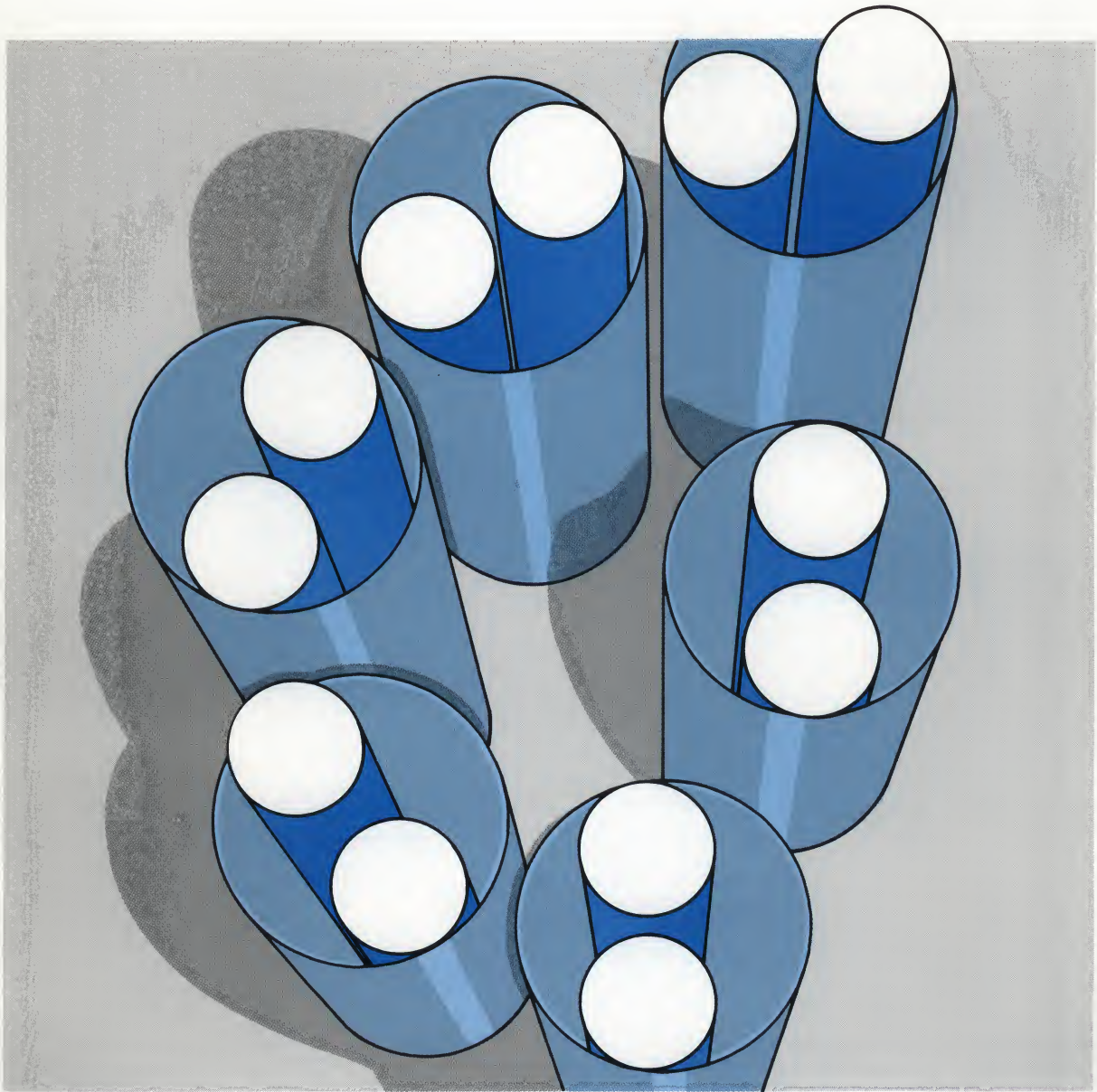
Figure 7-2. Additional Words Reserved by Some Compilers

The keywords **asm**, **fortran**, and **volatile** are reserved by the IBM RT PC C Language compiler. The IBM RT PC C Language compiler also reserves external identifiers beginning with `_C_` prefix. See *AIX Operating System Technical Reference* for information about any AIX Operating System functions that have names beginning with `_C_`.

Although names of system calls and library subroutines are not reserved words, do not use these names as identifiers. Duplication of a pre-defined name can lead to confusion for the maintainers of your code and to possible errors at link time. If you include a subroutine library in a program, be aware of the names of the subroutines in that library.

---

## Chapter 8. Data Definitions and Storage Classes



---

## CONTENTS

About This Chapter .....	8-3
Internal Data Definitions .....	8-4
External Data Definitions .....	8-6
Declarators .....	8-8
Initializers .....	8-12
<b>auto</b> Storage Class .....	8-14
<b>extern</b> Storage Class .....	8-19
<b>register</b> Storage Class .....	8-24
<b>static</b> Storage Class .....	8-27

---

## About This Chapter

This chapter describes C language variable definitions and the storage classes to which variables can be assigned. The location of a variable definition and storage class specified within the definition determine the following:

- Whether the variable is known throughout a file or only within the function or block where the variable is defined.
- Whether storage for the variable is maintained throughout program execution or only during the execution of the block where the variable is defined.
- Whether the variable is to be stored in memory or in a register.
- Whether the variable receives the default initial value **0** (zero) or an indeterminate default initial value.

For a function, storage class determines whether the function is known throughout the program or only within the source file where the function is defined.

---

# Internal Data Definitions

---

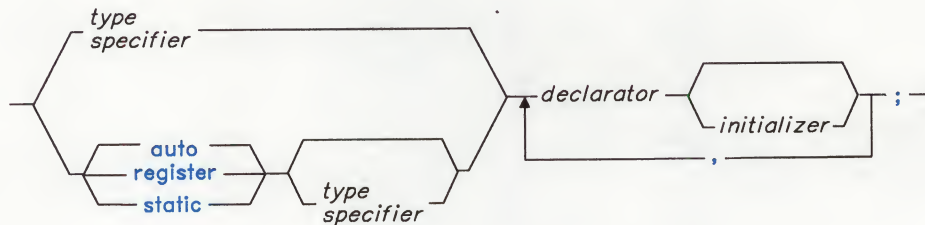
## Description

An *internal data definition* appears at the beginning of a block and:

- Describes a variable
- Causes the system to allocate storage for that variable
- Makes that variable accessible to the current block.

You can use internal data definitions to define variables that you want only the current block (and its nested blocks) to be able to access. An internal data definition has the form:

*internal data  
definition*



You can define an internal data object as having one of the following storage classes:

**auto**  
**register**  
**static**

If you do not specify a storage class in an internal data definition, all variables defined in that definition receive the storage class **auto**. If you specify a storage class, you can omit the type specifier. If you omit the type specifier, all variables defined in that definition receive the type **int**.

---

## Initialization

The types of variables you can initialize and the values that uninitialized variables receive vary for each storage class.

## Storage

The duration and type of storage varies for each storage class.

## Scope

You can use an internal variable only in the block in which you define the variable. If you place a block within this block, the inner block can use any variables defined in the outer block, provided the inner block does not define a variable having the same identifier.

## Related Information

“**auto** Storage Class” on page 8-14.

“**register** Storage Class” on page 8-24.

“**static** Storage Class” on page 8-27.

“Declarators” on page 8-8.

“Initializers” on page 8-12.

Chapter 10, “Primary Data Types.”

---

# External Data Definitions

---

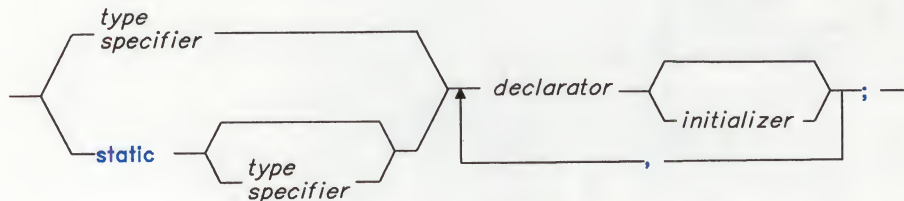
## Description

An **external data definition** appears outside a function definition and:

- Describes a variable
- Causes the system to allocate storage for that variable
- Makes that variable accessible to all functions that follow the definition and are located in the same file as the definition.

You can use an external data definition to define variables that you want several functions to be able to access. An external data definition has the form:

*external data  
definition*



The only storage class specifier you can place in an external data definition is **static**. However, if you do not specify **static**, all variables defined in that external data definition receive the storage class **extern**. If you specify the storage class **static**, you can omit the type specifier. If you omit the type specifier, all variables defined in that definition receive the type **int**.

---

## Initialization

You can initialize any external variable except a union. If you do not initialize an external variable, its initial value is 0 (zero). If you initialize an external variable, the initializer must be described by a constant expression or must reduce to the address of a previously defined variable, possibly modified by a constant expression. Initialization occurs at the start of program execution.

## Storage

The system allocates memory for all external variables when the program begins execution. The system frees this storage when the program finishes executing.

## Scope

You can use an external variable in any place that follows the definition and that is located in the same file as the definition. Also, you can use a variable that has the storage class **extern** outside of the file that contains the variable definition and in lines that precede the variable definition, provided that an **extern** declaration precedes the variable reference.

## Related Information

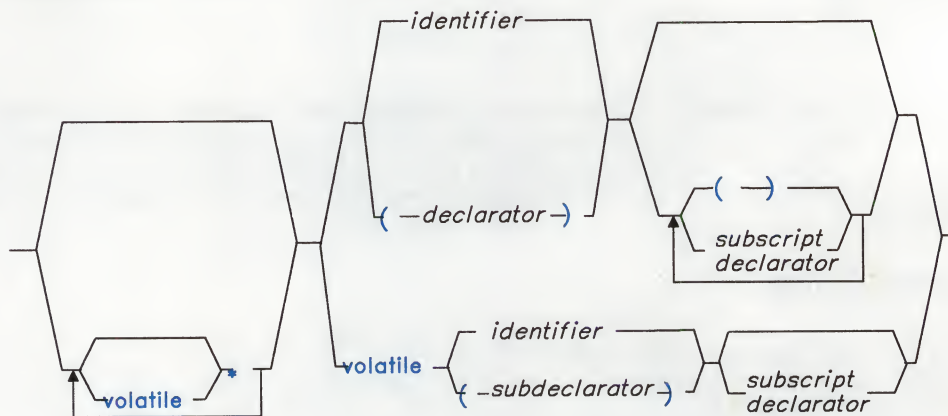
“**extern** Storage Class” on page 8-19.  
“**static** Storage Class” on page 8-27.  
“Declarators” on page 8-8.  
“Initializers” on page 8-12.  
Chapter 10, “Primary Data Types.”

## Declarators

## Description

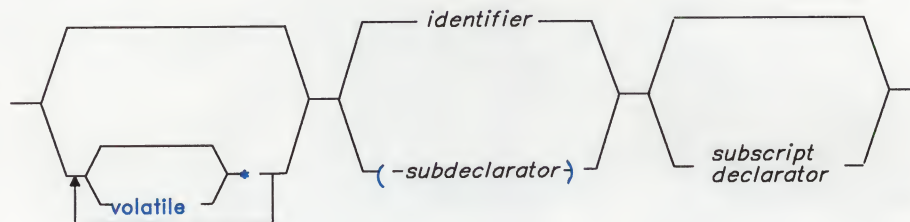
A **declarator** names a data object and further describes the type of data that the object represents. Declarators appear in all data definitions and declarations and in some type definitions. A declarator has the form:

*declarator*



A declarator cannot describe a function as having the **volatile** attribute. Thus, a declarator for a complex **volatile** data object can contain a *subdeclarator*. A subdeclarator has the form:

*subdeclarator*



---

A simple declarator consists of an identifier, which names a data object. For example, the following internal data definition uses `initial` as the declarator:

```
auto char initial;
```

The data object `initial` has the storage class **auto** and the data type **char**.

You can define or declare a complex data object by using a declarator that contains an identifier, which names the data object, and some combination of symbols and identifiers, which helps to describe the type of data that the object represents. The following declaration uses `compute( )` as the declarator:

```
extern long compute( );
```

The function `compute` is declared as having the **extern** storage class and returning a **long** value. "Examples" on page 8-10 provides more examples of declarators.

## volatile Attribute

The ***volatile attribute*** causes the compiler to place the value of the data object in storage and reload this value at each reference to the data object. The system does not optimize expressions that reference **volatile** data objects. The **volatile** attribute is useful for data objects having values that may be changed by low-level code.

You can give a data object the **volatile** attribute by placing the keyword **volatile** in the declarator. For a **volatile** pointer, you must place the keyword **volatile** between the **\*** and the identifier. For example:

```
int * volatile x;          /* x is a volatile pointer to an int */
```

For a pointer to a **volatile** data object, you must place the keyword **volatile** before the *\*identifier* sequence. For example:

```
int volatile *x;           /* x is a pointer to a volatile int */
```

---

For other types of **volatile** variables, the position of the keyword **volatile** within the definition (or declaration) is less important. For example:

```
volatile struct omega {  
    int limit;  
    char code;  
} group;  
  
struct omega test;
```

Provides the same storage as:

```
struct omega {  
    int limit;  
    char code;  
} volatile group;  
  
struct omega test;
```

In both examples, only the structure variable `group` receives the **volatile** attribute. Although enumeration, structure, and union variables can receive the **volatile** attribute, enumeration, structure, and union tags do not carry the **volatile** attribute. The keyword **volatile** cannot separate the keywords **enum**, **struct**, and **union** from their tags.

Functions cannot be defined or declared as having the **volatile** attribute.

## Examples

The following table describes some declarators:

Example	Description
<code>owner</code>	<code>owner</code> is a data object.
<code>*node</code>	<code>node</code> is a pointer to a data object.
<code>case[126]</code>	<code>case</code> is an array of 126 elements.
<code>*action( )</code>	<code>action</code> is a pointer to a function.
<code>volatile case</code>	<code>case</code> is a data object that has the <b>volatile</b> attribute.

Figure 8-1 (Part 1 of 2). Example Declarators

---

Example	Description
<code>* volatile volume</code>	volume is a <b>volatile</b> pointer to a data object.
<code>volatile * next</code>	next is a pointer to a <b>volatile</b> object.
<code>(* volatile points[3][2]) ( )</code>	points is a two-dimensional array of six <b>volatile</b> pointers to functions.
<code>volatile * sequence[5]</code>	sequence is an array of five pointers to <b>volatile</b> data objects.

---

**Figure 8-1 (Part 2 of 2). Example Declarators**

## Related Information

“Arrays” on page 11-8.

“Enumerations” on page 11-17.

“Pointers” on page 11-22.

“Structures” on page 11-30.

“Unions” on page 11-39.

“Defining New Data Types and New Names for Existing Data Types” on page 11-4.

“**volatile** Conversions” on page 14-10.

---

# Initializers

---

## Description

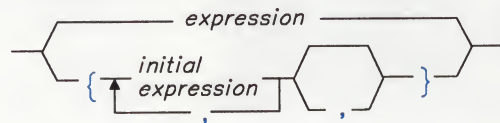
An **initializer** is an optional part of a data definition that specifies the original (or initial) value of a data object. An initializer has the form:

*initializer*

— = — *initial expression* —

The initializer consists of the = symbol followed by an **initial expression**. The initial expression evaluates to the original value of the data object. An initial expression has the form:

*initial expression*



You can use the simple initializer = *expression* to give a value to a primary data object (character, integer, or floating-point number) or to an enumeration variable or a pointer. For example, the following data definition uses the initializer = 3 to give the initial value 3:

```
register int group = 3;
```

For complex objects that represent multiple data objects (such as arrays and structures), the set of initial expressions must be enclosed in { } (braces). Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas. For example, the following definition initializes all 12 elements of the array grid:

---

```

static short grid[3] [4] =
{
    { 0, 0, 0, 1 },
    { 0, 0, 0, 1 },
    { 0, 0, 0, 1 }
}

```

The initial values of grid are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	0
grid[1] [1]	0	grid[1] [3]	1
grid[0] [2]	0	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	0	grid[2] [2]	0
grid[1] [1]	0	grid[2] [3]	1

Initialization considerations for each data type are described in the section for that data type.

## Related Information

- “Internal Data Definitions” on page 8-4.
- “External Data Definitions” on page 8-6.
- “Arrays” on page 11-8.
- “Characters” on page 10-4.
- “Enumerations” on page 11-17.
- “Floating-Point Variables” on page 10-6.
- “Integers” on page 10-8.
- “Pointers” on page 11-22.
- “Structures” on page 11-30.
- “Unions” on page 11-39.

---

## auto Storage Class

---

### Description

The **auto** storage class enables you to define variables and declare parameters whose usage is restricted to the current block. A variable having the **auto** storage class must be defined within a block or declared as a parameter to a function. The storage class keyword **auto** is optional in a definition and forbidden in a parameter declaration. If the following example lines were placed within a block, these lines would be **auto** variable definitions:

```
auto int counter;  
int x = 100;  
auto char letter = 'k';  
float number = 123.45;  
char names[10];  
int rate = interest + 1;
```

### Initialization

You can initialize any **auto** variable that is not an array, a structure, or a union (or a parameter). If you do not provide an initial value, the initial value is undefined. If you provide an initial value, the expression representing the initial value can be comprised of constants, previously defined variables, previously declared functions, and operators. Initialization occurs when the system allocates storage for the variable.

---

## Storage

The system allocates memory to an **auto** variable each time the block in which it is defined begins executing. When the block finishes executing, the system frees this memory. If an **auto** variable is defined within a block that is recursively invoked, the system allocates memory for the variable at each invocation of the block.

## Scope

You can use an **auto** variable only in the block in which you define the variable. If you place a block within this block, the inner block can use any of the variables defined in the outer block, provided the inner block does not define a variable having the same identifier.

## Usage

Defining variables as having the **auto** storage class may decrease the amount of memory required for program execution, because **auto** variables require storage only while they actually are needed.

Defining variables as having the **auto** storage class also may make code easier to maintain, because a change to an **auto** variable in one function rarely affects another function.

---

## Examples

The following program shows the scope and initialization of **auto** variables. `main` defines two different variables named `auto_var`. The first definition occurs on line 8. The second definition occurs in a nested block on line 11. While the nested block executes, only the `auto_var` created by the second definition is available. During the rest of the program, only the `auto_var` created by the first definition is available.

```
1  /* program to illustrate auto variables */
2
3  #include <stdio.h>
4
5  main()
6  {
7      void call_func();
8      int auto_var = 1;
9
10     {
11         int auto_var = 2;
12         printf("inner auto_var = %d\n", auto_var);
13     }
14     call_func(auto_var);
15     printf("outer auto_var = %d\n", auto_var);
16 }
17
18 void call_func(passed_var)
19 int passed_var;
20 {
21     printf("passed_var = %d\n", passed_var);
22     passed_var = 3;
23     printf("passed_var = %d\n", passed_var);
24 }
```

The previous program produces the following output:

```
inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1
```

---

The following example uses an array that has the storage class **auto** to pass a character string to the function `sort`. The C language views an array name that appears without subscripts (for example, `string`, instead of `string[0]`) as a pointer. Thus, `sort` receives the address of the character string, rather than the contents of the array. The address enables `sort` to change the values of the elements in the array.

```
/* Sorted string program */

#include <stdio.h>

main()
{
    void sort();
    char string[75];
    int length;

    printf("Enter letters:\n");
    scanf("%s", string);
    printf("Enter number of letters:\n");
    scanf("%d", &length);
    sort(string, length);
    printf("The sorted string is: %s\n", string);
}

void sort(array, n)
char array[ ];
int n;
{
    int gap, i, j, temp;

    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && array[j] > array[j + gap];
                j -= gap)
            {
                temp = array[j];
                array[j] = array[j + gap];
                array[j + gap] = temp;
            }
}
```

---

When run, interaction with the previous program could produce:

Output: Enter letters:  
Input: zyfab  
Output: Enter number of letters:  
Input: 5  
Output: The sorted string is: abfyz

## Related Information

“Internal Data Definitions” on page 8-4.

“Parameter Declaration” on page 12-7.

---

## extern Storage Class

---

### Description

The **extern** storage class enables you to define variables and functions that several source files can use. All variable declarations that occur outside a function and that do not contain a storage class specifier refer to **extern** variables. All function definitions that do not specify a storage class define **extern** functions.

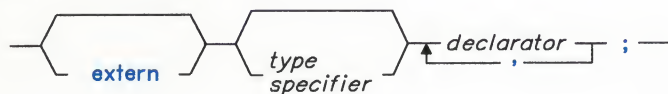
An **extern** variable definition allocates storage for the specified variable. An **extern** variable or function definition or declaration also makes the described variable or function usable by the succeeding part of the current source file. If you want to use an **extern** variable or function prior to its definition or in a file other than the file in which it is defined, you must explicitly declare the variable or function. This declaration does not replace the definition. The declaration just helps to describe the variable that is externally defined.

An **extern** declaration can be distinguished from an **extern** definition by the presence of the keyword **extern**. If the keyword **extern** is present, it is a declaration. Otherwise, it is a definition and a declaration. An **extern** definition can appear only outside a function definition. The IBM RT PC C Language compiler permits multiple **extern** declarations that include no definition; if necessary, storage for the variable involved is allocated at link time. Other compilers may require exactly one declaration of the variable without the keyword **extern**, and may take that declaration as a definition of the storage for the variable.

### Declaration

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration appears inside a block and describes a variable, the declaration must contain the keyword **extern**. If the declaration describes a function or appears outside a function and describes a variable, the keyword **extern** is optional. The declaration does not allocate storage. Therefore, it does not contain an initializer. An **extern** declaration has the form:

*extern  
declaration*



---

## Initialization of Variables

You can initialize any variable with the **extern** storage class except a **union**. If you initialize an **extern** variable, the initializer must appear in the variable definition and the initial value must be described by a constant expression or must reduce to the address of a previously defined variable, possibly modified by a constant expression. If you do not initialize an **extern** variable, its initial value is 0 (zero). Initialization occurs at the start of program execution.

## Storage for Variables

The system allocates memory for **extern** variables when the program begins execution. When the program finishes executing, the system frees this memory.

## Scope of Variables

You can use an **extern** variable in any place that follows the definition and that is located in the same source file as the definition. Also, you can use an **extern** variable any other place in the program, provided an **extern** declaration precedes the reference and this declaration is located in the same source file as the reference.

## Scope of Functions

You can use an **extern** function any place in the program. If the function has a return type other than `int` and the function is defined in another source file or later in the same source file, you should declare the function before calling the function.

---

## Examples

The following program shows the scope of **extern** variables and functions. The **extern** variable `total` is defined on line 18. The **extern** function `tally` is defined on lines 20 through 28. (The function `tally` can be placed in the same file as `main` or in a different file.) Because `main` precedes these definitions and `main` uses both `total` and `tally`, `main` declares `total` on line 10 and `tally` on line 11.

```
1  /*
2  ** This program receives the price of an item, adds a five percent
3  ** tax to the item, and prints the total cost of the item.
4  */
5
6  #include <stdio.h>
7
8  main()
9  {   /* begin main */
10     extern float total;
11     void tally();
12
13     printf("Enter the purchase amount: \n");
14     tally();
15     printf("\nWith tax, the total is: %.2f\n", total);
16 }   /* end main   */
17
18 float total;
19
20 void tally()
21 {   /* begin tally */
22     float tax, tax_rate = .05;
23
24
25     scanf("%f", &total);
26     tax = tax_rate * total;
27     total += tax;
28 }   /* end tally   */
```

---

The following program shows **extern** variables used by two functions. Because both functions `main` and `sort` can access and change the values of the **extern** variables `string` and `length`, `main` does not have to pass parameters to `sort`.

```
/* Sorted string program */

#include <stdio.h>

char string[75];
int length;

main()
{
    void sort();

    printf("Enter letters:\n");
    scanf("%s", string);
    printf("Enter number of letters:\n");
    scanf("%d", &length);
    sort();
}

void sort()
{
    int gap, i, j, temp;

    for (gap = length / 2; gap > 0; gap /= 2)
        for (i = gap; i < length; i++)
            for (j = i - gap; j >= 0 && string[j] >
                string[j + gap]; j -= gap)
            {
                temp = string[j];
                string[j] = string[j + gap];
                string[j + gap] = temp;
            }
    printf("The sorted string is: %s\n", string);
}
```

---

When run, interaction with the previous program could produce:

Output: Enter letters:  
Input: zyfab  
Output: Enter number of letters:  
Input: 5  
Output: The sorted string is: abfyz

## Related Information

“External Data Definitions” on page 8-6.  
“Function Definition” on page 12-5.  
“Function Declarator” on page 12-6.  
“Constant Expression” on page 13-9.

---

## register Storage Class

---

### Description

The **register** storage class enables you to indicate within an internal data definition or a parameter declaration that the variable being described will be heavily used (such as a loop control variable). A variable having the **register** storage class must be defined within a block or declared as a parameter to a function. The storage class keyword **register** is required in a data definition and a parameter declaration that describes a variable having the **register** storage class. The following example lines define variables (or declare parameters, depending on the location of the lines) having the **register** storage class:

```
register int score1 = 0, score2 = 0;  
register unsigned char code = 'A';  
register int *element = &order[0];
```

### Initialization

You can initialize any **register** variable that is not an array, a structure, or a union (or a parameter). If you do not provide an initial value, the initial value is undefined. If you provide an initial value, the expression representing the initial value can be comprised of constants, previously defined variables, previously declared functions, and operators. Initialization occurs when the system allocates storage for the variable.

### Storage

The system allocates resources for a **register** variable each time the block in which the variable is defined begins executing. When the block finishes executing, the system frees these resources.

The **register** storage class indicates that a variable is heavily used and suggests to the compiler that the value of the variable reside in a register. Not all **register** variables are placed in registers. Most compilers allocate registers for only certain types of variables. Furthermore, most systems have a limited number of registers available for **register** variables.

The IBM RT PC C Language compiler can allocate registers for any arithmetic or pointer variable. The IBM RT PC C Language compiler can assign up to seven registers to integral and pointer variables and four registers to floating-point variables.

---

If the compiler cannot allocate a register for a **register** variable, the compiler treats the variable as having storage class **auto**. Because of the limited size and number of registers available on most systems, few variables can be stored in registers at the same time.

## Scope

You can use a **register** variable only in the block in which you define the variable. If you place a block within this block, the inner block can use any of the variables defined in the outer block, provided the inner block does not define a variable having the same identifier.

## Restrictions

You cannot apply the **&** (address) operator to register variables.

## Usage

Using register definitions for variables that are heavily used may make your object files appear smaller and run faster. In object code, a reference to a register usually requires less code and time than a reference to memory.

## Examples

The following two programs count to one million. The first program defines **ones** and **thousands** as **auto** variables. The second program defines **ones** and **thousands** as **register** variables. On most machines the **register** variable definitions cause the second program to run faster than the first program.

```
/* slow */
main()
{
    int ones;
    int thousands;

    for (thousands = 0; thousands <= 1000; ++thousands)
    {
        for (ones = 0; ones <= 1000; ++ones)
        {
            ;
        }
    }
}
```

---

```
/* probably faster */
main()
{
    register int ones;          /* use a register      */
    register int thousands;     /* use another register */

    for (thousands = 0; thousands <= 1000; ++thousands)
    {
        for (ones = 0; ones <= 1000; ++ones)
        {
            ;
        }
    }

    /* the registers are freed */
}
```

## Related Information

“Internal Data Definitions” on page 8-4.  
“Parameter Declaration” on page 12-7.  
“Address” on page 13-19.

---

## static Storage Class

---

### Description

The **static** storage class enables you to define variables that retain storage throughout program execution and can be used by only one block or a limited set of functions in one source file. The **static** storage class also enables you to define functions that can be called by other functions in the same source file. The keyword **static** must appear in all **static** variable definitions, and **static** function definitions.

A variable having the **static** storage class can be defined within a block or outside of a function. If the definition occurs within a block, the variable is an internal variable. If the definition occurs outside of a function, the variable is an external variable.

### Initialization of Variables

You can initialize any **static** variable that is not a union. If you do not provide an initial value, the variable receives the value 0 (zero). If you initialize a **static** variable, the initializer must be described by a constant expression or must reduce to the address of a previously defined variable, possibly modified by a constant expression. Initialization occurs at the start of program execution.

### Storage for Variables

The system allocates memory for a **static** variable when the program begins execution. When the program finishes executing, the system frees this memory.

---

## Scope of Variables and Functions

You can use an internal **static** variable only in the block in which you define the variable. If you place a block within this block, the inner block can use any of the variables defined in the outer block, provided the inner block does not define a variable having the same identifier.

You can use an external **static** variable or a **static** function any place following the definition in the source file that contains the definition.

## Usage

Following are some reasons for using internal **static** variables:

- You need a variable for which storage remains allocated throughout program execution. For example, the variable `counter`, which keeps track of how many times a function is called:

```
static int counter = 0;
```

- You need a variable that is initialized to a value that never changes. Using the **static** storage class keeps the system from re-initializing the variable each time the block in which the variable is defined executes. For example:

```
static float rate = 10.5;
```

- You need an array that is initialized. (You cannot initialize an array that has the **auto** storage class.) For example:

```
static char message[ ] = "startup completed";  
static int integers[10] =  
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

---

## Examples

The following program shows the scope of external **static** variables. This program uses two different external **static** variables named `stat_var`. The first definition occurs in file 1. The second definition occurs in file 2. The `main` function references the variable defined in file 1. The `var_print` function references the variable defined in file 2:

```
/* program to illustrate external static variables */
```

```
/* file 1 */
```

```
extern void var_print();  
static stat_var = 1;
```

```
main()
```

```
{  
    printf("file1 stat_var = %d\n", stat_var);  
    var_print();  
}
```

```
/*
```

```
** The following text must be located in a  
** different file than the preceding text.
```

```
*/
```

```
/* file 2 */
```

```
static int stat_var = 2;
```

```
void var_print()
```

```
{  
    printf("file2 stat_var = %d\n", stat_var);  
}
```

The preceding program produces the following output:

```
file1 stat_var = 1  
file2 stat_var = 2
```

---

The following program shows the scope of internal **static** variables. The function `test` defines the **static** variable `stat_var`. `stat_var` maintains storage throughout the program, even though `test` is the only function that can reference `stat_var`.

```
/* program to illustrate internal static variables */
main()
{
    void test();
    int counter;
    for (counter = 1; counter <= 4; ++counter)
    {
        test();
    }
}

void test()
{
    static int stat_var = 0;
    auto int auto_var = 0;

    stat_var++;
    auto_var++;
    printf("stat_var = %d auto_var = %d\n", stat_var, auto_var);
}
```

The preceding program produces the following output:

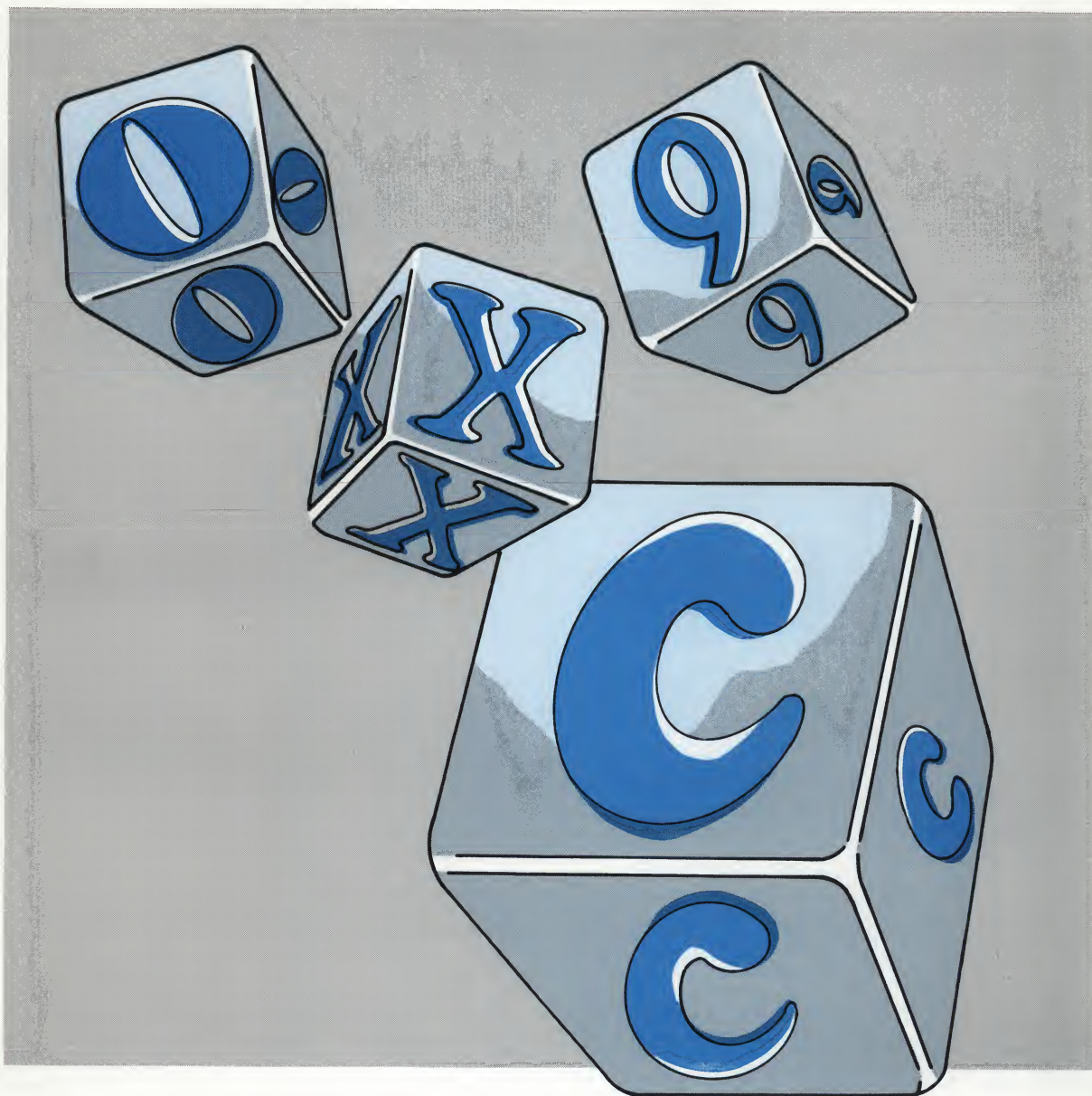
```
stat_var = 1 auto_var = 1
stat_var = 2 auto_var = 1
stat_var = 3 auto_var = 1
stat_var = 4 auto_var = 1
```

## Related Information

- “Internal Data Definitions” on page 8-4.
- “External Data Definitions” on page 8-6.
- “Function Definition” on page 12-5.
- “Function Declarator” on page 12-6.

---

## Chapter 9. Constants



---

## CONTENTS

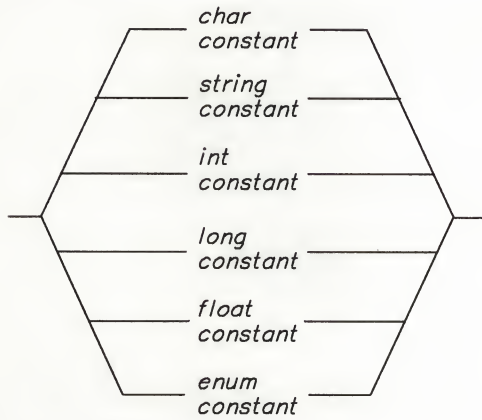
About This Chapter .....	9-3
Character .....	9-4
Decimal .....	9-6
Enumeration .....	9-8
Escape Sequence .....	9-10
Float .....	9-12
Hexadecimal .....	9-15
Integer .....	9-17
Long .....	9-18
Octal .....	9-19
String .....	9-21

---

## About This Chapter

This chapter describes the C language constants. The C language contains the following types of constants:

*constant*



Typically, a constant is a data object with a value that does not change during the execution of a program. You can, however, change the value of a string. Nevertheless, strings are grouped with the C language constants.

---

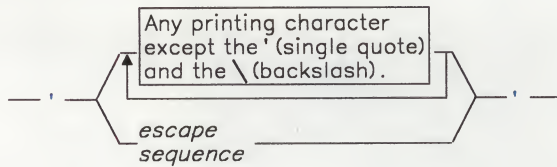
# Character

---

## Definition

A **character constant** contains a character or an escape sequence enclosed in single quotation symbols. A character constant has the form:

*char  
constant*



Some compilers allow you to place more than one character or escape sequence in a character constant. The IBM RT PC C Language compiler allows you to place one to four characters in a character constant. The character constant, however, can occupy no more than four bytes of storage. Thus, although you can place four 1-byte characters in a character constant, you can place only two 2-byte characters in a character constant.

## Value

The value of a character constant is the numeric representation of the character in the character set of the system that runs the program.

## Data Type

A character constant has an **int** type. The IBM RT PC C Language compiler treats character constants as having **unsigned int** type.

---

## Restrictions

The character must appear in the character set on the system that runs the program.

You cannot use the new-line character in a character constant. However, you can represent the new-line character by the `\n` new-line escape sequence.

You can represent the double quotation symbol by itself, but you must use the `\'` escape sequence to represent the single quotation symbol.

## Examples

`'a'`  
`'$'`  
`'7'`

`'\''`  
`'x'`  
`'\117'`

`'0'`  
`'\n'`  
`'C'`

## Related Information

“String” on page 9-21.

“Escape Sequence” on page 9-10.

“Integers” on page 10-8.

---

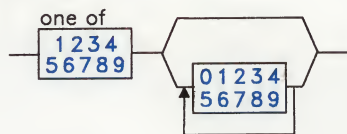
# Decimal

---

## Definition

A **decimal constant** contains any of the digits **0** (zero) through **9**. The first digit cannot be **0** (zero). A decimal constant has the form:

*decimal  
constant*



The compiler interprets the integer **0** as an octal constant, rather than as a decimal constant.

## Data Type

A decimal constant with a value that exceeds the largest **int** value allowed on the system has **long int** type. A smaller decimal constant has **int** type.

A decimal constant always has a non-negative value, except when the value provided for the decimal constant is larger than the largest value allowed for a **long int**. A **-** symbol can precede a decimal constant. However, the C language treats the **-** symbol as a unary operator rather than as part of the decimal constant value.

The range of **int** values allowed by the IBM RT PC C Language compiler is -2,147,483,648 to 2,147,483,647. The range of **long int** values is the same as the range of **int** values for the IBM RT PC C Language compiler.

---

## Examples

485976  
433132211  
20  
5

## Related Information

“Octal” on page 9-19.  
“Hexadecimal” on page 9-15.  
“Integer” on page 9-17.  
“Integers” on page 10-8.

---

# Enumeration

---

## Definition

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an **enumeration constant**. An enumeration constant has the form:

*enum*  
*constant*  
— *identifier* —

## Value

Each enumeration constant has an integer value. You can assign a value by placing an = symbol and a constant expression after the enumeration constant.

If you do not assign values, the leftmost constant receives the value 0 (zero). Reading from left to right, each constant value increases by one.

When you refer to an enumeration constant, you can use the identifier or the integer value.

## Data Type

An enumeration constant has **int** type.

---

## Examples

The following data type definitions list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };  
           0      1      2      3      4
```

```
enum grain { oats=1, wheat, barley, corn, rice };  
           1      2      3      4      5
```

```
enum grain { oats, wheat=10, barley, corn=20, rice };  
           0      10      11      20      21
```

## Related Information

“Enumerations” on page 11-17.

“Defining New Data Types and New Names for Existing Data Types” on page 11-4.

“Integers” on page 10-8.

---

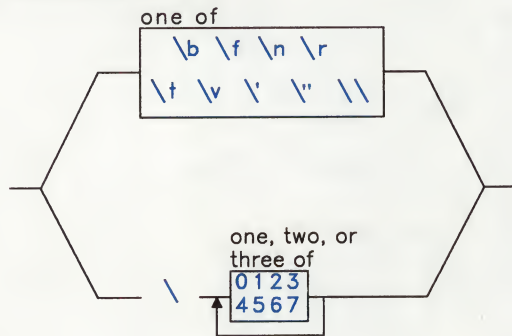
# Escape Sequence

---

## Definition

You can represent any member of the system character set by an **escape sequence**. You can use escape sequences to place such characters as tab, carriage return, and backspace into an output stream. An escape sequence contains a `\` symbol followed by one of the characters: **b**, **f**, **n**, **r**, **t**, **v**, **'**, **"**, or **\** or followed by one to three octal digits. An escape sequence has the form:

*escape  
sequence*



## Value

The value of an escape sequence is the member of the system character set that the escape sequence represents. For example, on a system that uses the ASCII character codes, the value of the escape sequence `\166` is the letter `v`. The RT PC system character set uses ASCII character codes to represent characters.

## Data Type

An escape sequence does not have a data type. You can place an escape sequence in a character constant or in a string constant.

---

## Notes

If the system does not recognize a \ and its following character or digits as the representation of a member of the system character set, the interpretation of the escape sequence is undefined.

When you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a \\ backslash escape sequence.

## Examples

Some C language escape sequences and the characters they represent are:

---

Escape Sequence	Character Represented
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation
\"	Double quotation
\\	Backslash
\0	NUL character (used in the C language to indicate the end of a character string)
\new-line character	Line continuation character (used in C language character strings to indicate the current line continues on the next line)
\147	ASCII code for the letter g

---

**Figure 9-1. Escape sequences**

## Related Information

“Character” on page 9-4.

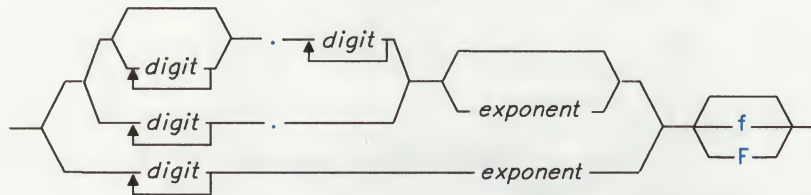
“String” on page 9-21.

Appendix F, “RT PC Character Codes.”

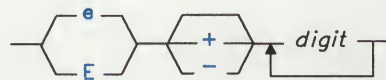
### Definition

The IBM RT PC C Language compiler allows you to modify a float constant by placing an **f** or **F** at the end of the constant. The **f** or **F** causes the constant to have type **float**. Thus, the constant is interpreted as a single precision value and is not promoted to type **double**. Some C compilers do not recognize this construct and interpret the **f** or **F** as an error.

float  
constant



*exponent*



---

## Value

The float constant 8.45e+3 evaluates as follows:

$$8.45 * 10^3 = 8450$$

The C language does not specify how float constants are represented in the system. If a float constant is too large or too small, the result is undefined.

The IBM RT PC C Language compiler supports the floating-point format defined by the ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. When you write a floating-point value to a display or a printer, a special value may appear in place of the floating-point number.

## Data Type

A float constant that ends with an **f** or **F** has type **float**. A float constant that does not contain this suffix has type **double**.

A float constant always has a non-negative value, except when the value provided for the float constant is larger than the largest value allowed by its type. A - symbol can precede a float constant. However, the C language treats the - symbol as a unary operator rather than as part of the float constant value.

## Examples

---

Float Constant	Value
5.3876e4	53,876
4e-11	.000000000004
1e+5	100,000
7.321E-3	.007321
3.2E+4	32,000
.5e-6	.0000005
.45	.45
6.e10	60,000,000,000
9.2e-2f	.092
8.9837E+2F	898.37

---

**Note:** When you use the **printf** function to display a float constant value, make certain the **printf** conversion code modifiers that you specify are large enough for the float constant value.

---

## Related Information

“Floating-Point Variables” on page 10-6.

“Floating-Point Representation” on page A-2.

---

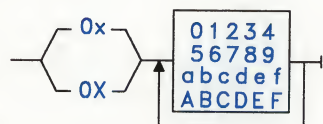
# Hexadecimal

---

## Definition

A **hexadecimal constant** begins with the **0** (zero) digit followed by the **x** or **X** letter. After the **0x** or **0X**, you can place any combination of the digits **0** (zero) through **9** and the letters **a** through **f** or **A** through **F**. A hexadecimal constant has the form:

*hexadecimal  
constant*



## Data Type

A hexadecimal constant with a value that exceeds the largest **unsigned int** value allowed on the system has type **long int**. A smaller hexadecimal constant has type **int**.

A hexadecimal constant always has a non-negative value, except when the value provided for the hexadecimal constant is larger than the largest value allowed for a **long int**. A **-** symbol can precede a hexadecimal constant. However, the C language treats the **-** symbol as a unary operator rather than as part of the hexadecimal constant value.

The range of **unsigned int** values allowed by the IBM RT PC C Language compiler is 0 to 4,294,967,295. The range of **int** and **long int** values for the IBM RT PC C Language compiler is the same: -2,147,483,648 to 2,147,483,647.

---

## Examples

0x3b24  
0XF96  
0x21  
0x3AA  
0X29b  
0X4bD

## Related Information

“Integer” on page 9-17.  
“Decimal” on page 9-6.  
“Octal” on page 9-19.  
“Integers” on page 10-8.

---

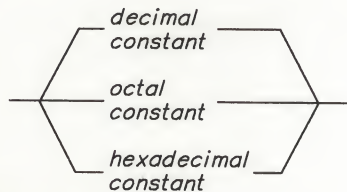
# Integer

---

## Definition

The C language contains three types of ***integer constants***. The following diagram lists these types:

*int*  
*constant*



## Data Type

A decimal constant with a value that exceeds the largest **int** and a hexadecimal or octal constant that exceeds the largest **unsigned int** allowed on your system has type **long int**. Other integer constants have type **int**.

The range of **unsigned int** values allowed by the IBM RT PC C Language compiler is 0 to 4,294,967,295. The range of **int** and **long int** values for the IBM RT PC C Language compiler is the same: -2,147,483,648 to 2,147,483,647.

## Related Information

"Decimal" on page 9-6.

"Octal" on page 9-19.

"Hexadecimal" on page 9-15.

"Integers" on page 10-8.

---

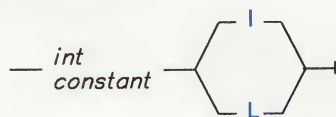
# Long

---

## Definition

A **long constant** contains a decimal, octal, or hexadecimal constant followed by an uppercase or lowercase letter **L**. A long constant has the form:

*long  
constant*



## Data Type

A long constant has type **long int**.

## Examples

035L  
4105l  
0x69a1L  
3521L  
04152l  
0x450694l

## Related Information

“Decimal” on page 9-6.  
“Octal” on page 9-19.  
“Hexadecimal” on page 9-15.  
“Integers” on page 10-8.

---

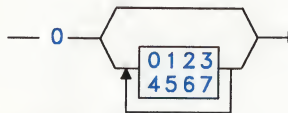
# Octal

---

## Definition

An **octal constant** begins with the digit **0** (zero) and contains any of the digits **0** (zero) through **7**. An octal constant has the form:

*octal  
constant*



## Data Type

An octal constant with a value that exceeds the largest **unsigned int** allowed on your system has type **long int**. A shorter octal constant has type **int**.

An octal constant always has a non-negative value, except when the value provided for the octal constant is larger than the largest value allowed for a **long int**. A **-** symbol can precede an octal constant. However, the C language treats the **-** symbol as a unary operator rather than as part of the octal constant value.

The range of **unsigned int** values allowed by the IBM RT PC C Language compiler is 0 to 4,294,967,295. The range of **int** and **long int** values for the IBM RT PC C Language compiler is the same: -2,147,483,648 to 2,147,483,647.

---

## Examples

0  
0125  
034673  
03245

## Related Information

“Integer” on page 9-17.  
“Decimal” on page 9-6.  
“Hexadecimal” on page 9-15.  
“Integers” on page 10-8.

---

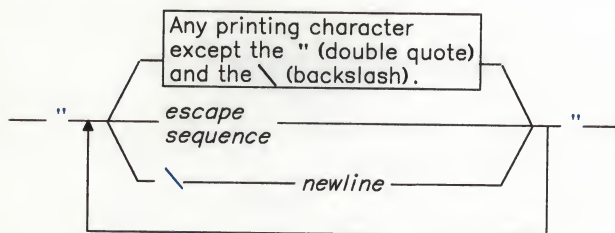
# String

---

## Definition

A **string constant** contains a sequence of characters enclosed in double quotation symbols. A string constant has the form:

*string  
constant*



## Value

The compiler places a `\0` (NUL) character at the end of each string. By convention, programs recognize the end of a string by finding the NUL character.

All string constants are distinct, even if their values are equivalent. Thus, if the string "abcd" appears three times in a program, the compiler reserves storage for three data objects that have the constant value "abcd".

Unlike other constants, you can change the value of a string. By defining a pointer to the string, you can access and change the contents of the string.

## Data Type

A string constant has type *array of char* and storage class **static**.

---

## Restrictions

If you want to continue a string on the next line, you must place the `\` symbol at the end of the line to be continued. The compiler reads this backslash as a line continuation symbol, rather than as a character in the string.

In a string, a new-line character can appear following the line continuation symbol only. You can, however, use the escape sequence `\n` to represent a new-line character as part of the string.

You can represent the single quotation symbol by itself `'`, but you use the escape sequence `\'` to represent the double quotation symbol.

## Examples

```
"Bach's \"Jesu, Joy of Man's Desiring\""
```

```
"A string has the form: "
```

```
"The following tests proved positive:\n"
```

```
""
```

```
"Last Name
```

```
First Name
```

```
MI
```

```
Street Address
```

```
City
```

```
State
```

```
Zipcode "
```

## Related Information

“Character” on page 9-4.

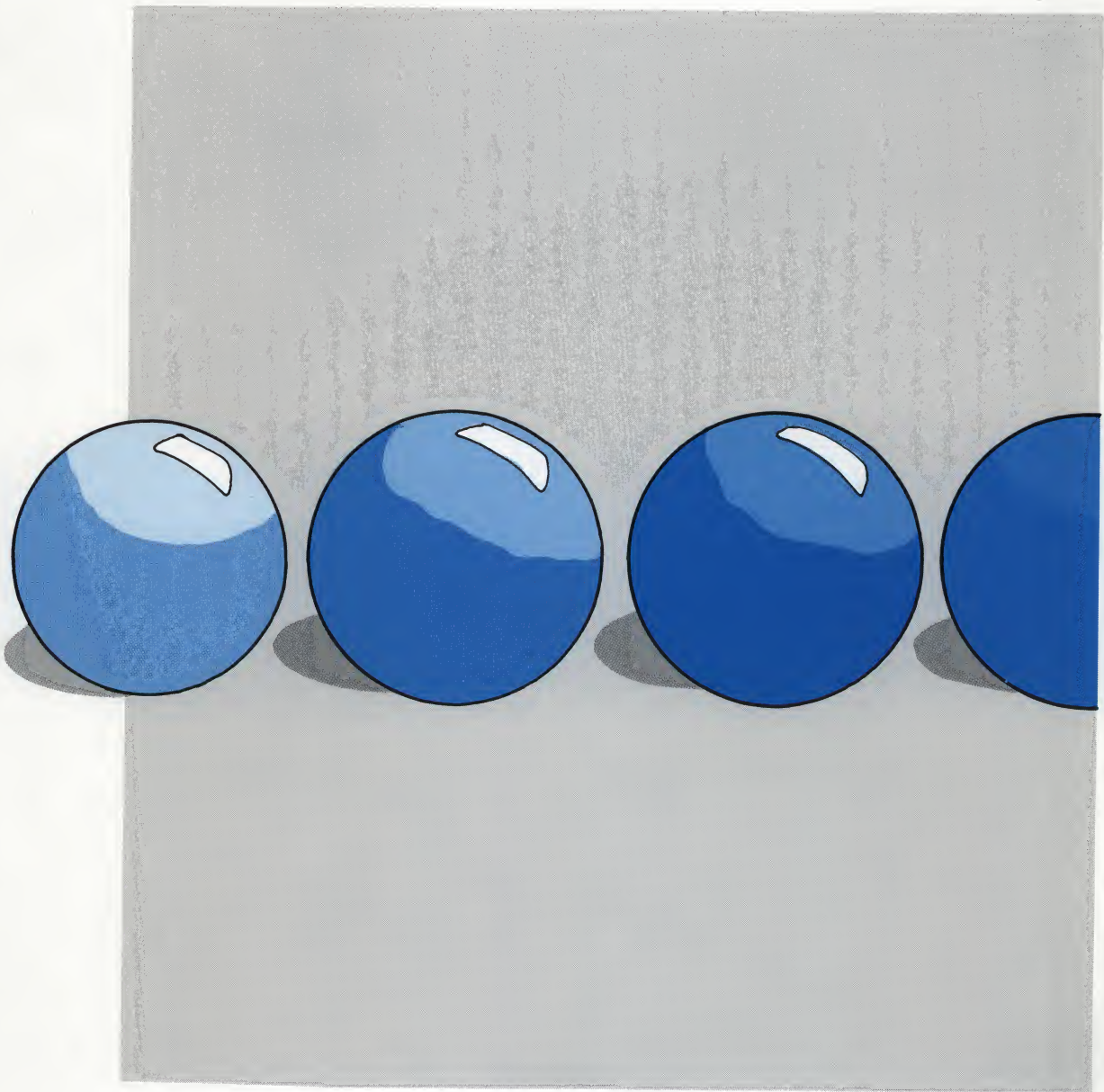
“Escape Sequence” on page 9-10.

“Characters” on page 10-4.

“Arrays” on page 11-8.

---

## Chapter 10. Primary Data Types



---

## CONTENTS

About This Chapter .....	10-3
Characters .....	10-4
Floating-Point Variables .....	10-6
Integers .....	10-8
<b>void</b> Type .....	10-10

---

## About This Chapter

This chapter describes the primary data types:

- Characters
- Floating-Point Numbers
- Integers
- **void** Functions.

From these types, you can derive the following types:

- Arrays
- Pointers
- Structures
- Unions
- Functions.

Chapter 12, “Functions” describes functions and Chapter 11, “Complex Data Types” describes the other listed data types as well as enumeration data types.

---

# Characters

---

## Description

The C language contains two character data types: **char** and **unsigned char**. These data types provide enough storage to hold any member of the system character set. The IBM RT PC C Language compiler treats both the **char** and the **unsigned char** data types as **unsigned** values.

The value of a character variable is the non-negative integer code for the character that the variable represents. You can store other quantities as character variables, but the C language does not define how these quantities are treated.

You can use a character variable like an integer variable. Some systems even treat a **char** variable as a negative quantity when its high-order bit is set. If you define a character variable as having type **unsigned char**, any system will treat the value of the variable as a non-negative quantity.

To define or declare a data object having a character data type, place a **char specifier** in the type specifier position of the definition or declaration. The **char** specifier has the form:

*char  
specifier*



The declarator for a simple character definition or declaration is an identifier. You can initialize a simple character with a character constant or with an expression that evaluates to an integer. (The storage class of a variable determines how you can initialize the variable.)

---

## Examples

The following example defines the **char** variable `end_of_string` as having the initial value `\0` (the null character):

```
char end_of_string = '\0';
```

The following example defines the **unsigned char** variable `switches` as having the initial value 3:

```
unsigned char switches = 3;
```

You can use the **char** specifier in variable definitions to define such complex variables as: arrays of characters, pointers to characters, and arrays of pointers to characters.

The following example defines `string_pointer` as a pointer to a character:

```
char *string_pointer;
```

The following example defines `name` as a pointer to a character. Initially, `name` points to the first letter in the character string "Johnny":

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are: a pointer to the string "Venus", a pointer to "Jupiter", and a pointer to "Saturn":

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

## Related Information

"Arrays" on page 11-8

"Pointers" on page 11-22

"Character" on page 9-4.

"Assignment Expression" on page 13-33.

---

# Floating-Point Variables

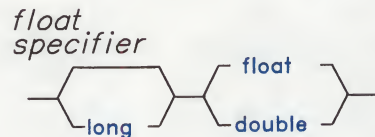
---

## Description

The C language contains two types of floating-point variables: **float** (single precision) and **double** or **long float** (double precision). (The keyword **double** is more commonly used than **long float**.) Some compilers support only one size of floating-point objects. Although these compilers treat **float** and **double** as the same precision, the C language still treats **float** and **double** as distinct data types.

The IBM RT PC C Language compiler recognizes an additional type of floating-point variable, **long double**. Compilers that implement this type, provide **long double** variables with the same or greater precision as **double** variables. The IBM RT PC C Language compiler gives **float** objects a 32-bit size and gives **double** and **long double** objects a 64-bit size.

To define or declare a data object having a floating-point quantity, use the **float specifier**. The IBM RT PC C Language float specifier has the form:



The declarator for a simple floating-point definition or declaration is an identifier. You can initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. (The storage class of a variable determines how you can initialize the variable.)

The IBM RT PC C Language compiler supports the floating-point format defined by the ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. When you write a floating-point value to a display or a printer, a special value may appear in place of the floating-point number.

---

## Examples

The following example defines the **double** variable `pi`:

```
double pi;
```

The following example defines the **float** variable `real_number` as having the initial value 100.55:

```
static float real_number = 100.55;
```

The following example defines the **float** variable `float_var` as having the initial value 0.0143:

```
float float_var = 1.43e-2;
```

The following example declares the **long double** variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` as having 20 values of type **double**:

```
double table[20];
```

## Related Information

“Float” on page 9-12.

“Assignment Expression” on page 13-33.

“Integers” on page 10-8.

“Floating-Point Representation” on page A-2.

---

# Integers

---

## Description

The C language contains the following six types of integer variables:

- **short int** or **short**
- **int** (or by providing no type specifier; see “Internal Data Definitions” on page 8-4 and “External Data Definitions” on page 8-6)
- **long int** or **long**
- **unsigned short int** or **unsigned short**
- **unsigned** or **unsigned int**
- **unsigned long int** or **unsigned long**.

The C language lets each compiler determine the size of each integer data type. Some compilers provide **short**, **int**, and **long** variables with the same amount of storage. Other compilers provide **short** and **int** variables (or **long** and **int** variables) with the same amount of storage. The C language requires only that the storage size of a **short** variable is less than or equal to the storage size of an **int** variable and that the storage size of an **int** variable is less than or equal to the storage size of a **long** variable. Thus, the following expression always evaluates to 1 (true):

```
sizeof(short) <= sizeof(int) && sizeof(int) <= sizeof(long)
```

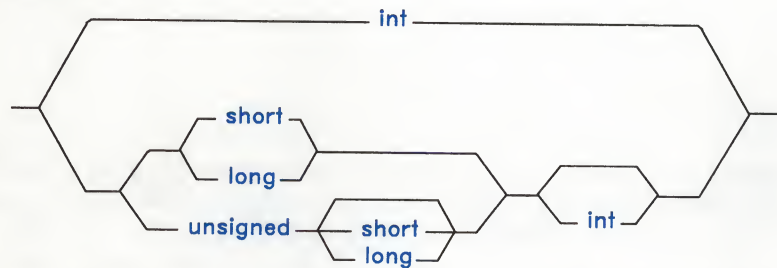
The **int** size usually represents the most efficient data storage size on the system (the word-size of the machine).

The IBM RT PC C Language compiler provides two sizes for integer data types. Objects having size **short** receive 16 bits of storage. Objects having size **int** or **long** receive 32 bits of storage.

The **unsigned** prefix makes the compiler treat a variable as a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, **int** reserves the same storage as **unsigned int**. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

To define or declare a data object having an integer data type, place an **int specifier** in the type specifier position of the definition or declaration. The **int** specifier has the form:

*int  
specifier*



The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to an integer. (The storage class of a variable determines how you can initialize the variable.)

## Examples

The following example defines the **short int** variable `flag`:

```
short int flag;
```

The following example defines the **int** variable `result`:

```
int result;
```

The following example defines the **unsigned long int** variable `ss_number` as having the initial value 438888834:

```
unsigned long ss_number = 438888834;
```

The following example defines the **int** variable `sum`. The initial value of `sum` is the result of the expression `a + b`:

```
auto sum = a + b;
```

## Related Information

“Integer” on page 9-17.

“Decimal” on page 9-6.

“Octal” on page 9-19.

“Hexadecimal” on page 9-15.

---

## void Type

---

### Description

The C language provides a data type that always represents an empty set of values. The keyword for this type is **void**. When a function does not return a value, you can use **void** as the type specifier in the function definition and declaration.

The C language does not require that a function returning no value be defined and declared as **void**. The **lint** program checker, however, produces warning messages for each function that does not return the same type of value as is specified in the function definition, in each declaration, or in each call to the function.

You cannot place a call to a function that has type **void** in any context that requires the call to yield a value.

### Examples

On line 3 of the following example, the function `find_max` is declared as having type **void**. On line 11, `find_max` is defined as having type **void**. Lines 11 through 22 contain the complete definition of `find_max`.

```
1  #include <stdio.h>
2
3  extern void find_max();
4
5  main()
6  {
7      static int numbers[ ] = { 99, 54, -102, 89 };
8
9      find_max(numbers);
10 }
```

```

11 void find_max(x)
12 int x[ ];
13 {
14     int i, temp = x[0];
15
16     for (i = 1; i < 4; i++)
17     {
18         if (x[i] > temp)
19             temp = x[i];
20     }
21     printf("max number = %d\n", temp);
22 }

```

On line 7 of the following example, main calls the function find\_max. Because find\_max returns a value and main does not want to receive that value, line 7 uses the cast operator (void) to convert the type of the return value to **void**. Thus, the return value of find\_max is discarded.

```

1  #include <stdio.h>
2
3  main()
4  {
5      static int numbers[ ] = { 99, 54, -102, 89 };
6
7      (void) find_max(numbers);
8  }
9
10 int find_max(x)
11 int x[ ];
12 {
13     int i, temp = x[0];
14
15     for (i = 1; i < 4; i++)
16     {
17         if (x[i] > temp)
18             temp = x[i];
19     }
20     return(temp);
21 }

```

---

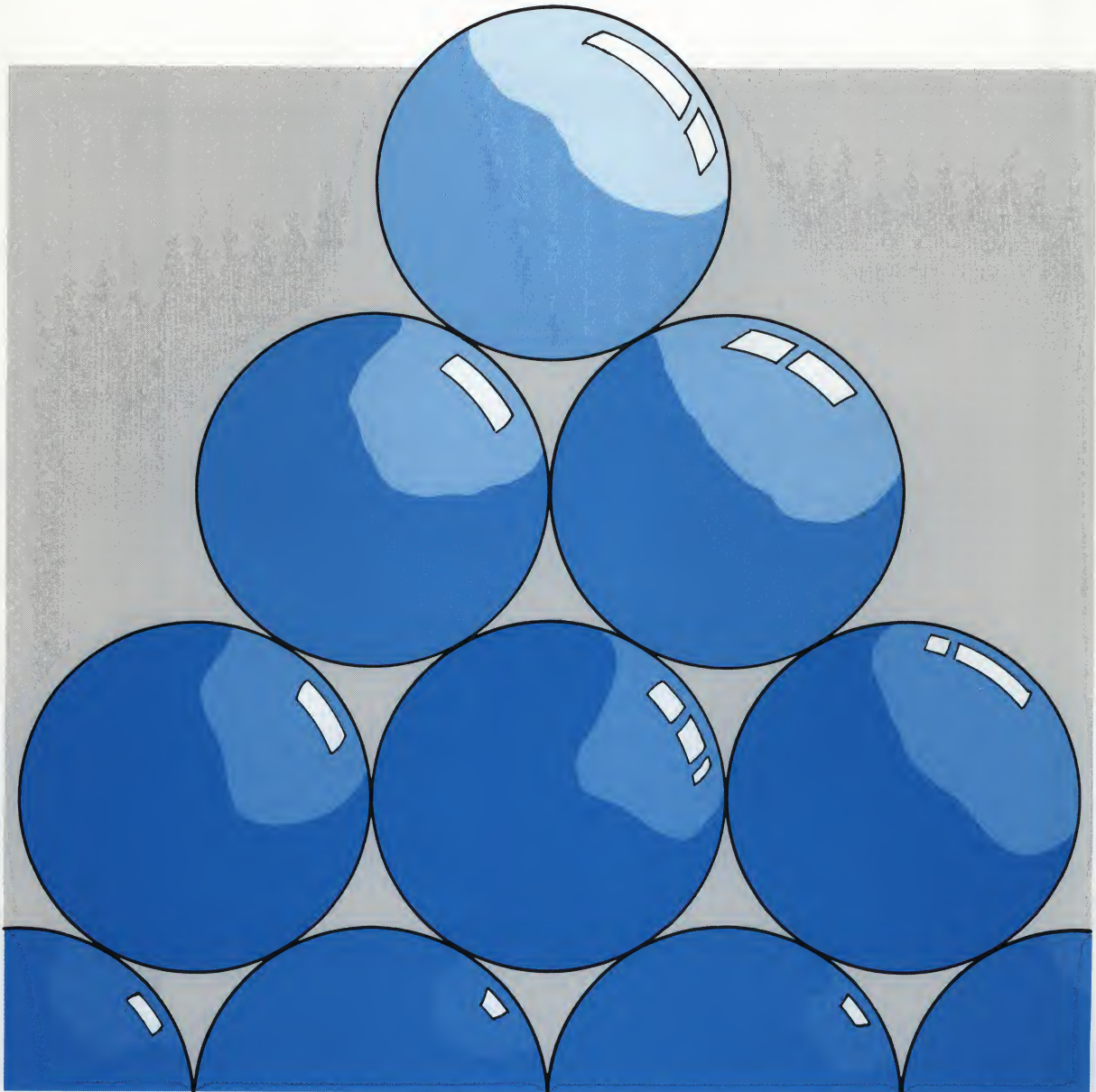
## Related Information

“void Conversions” on page 14-10.

“Cast” on page 13-20.

---

## Chapter 11. Complex Data Types



---

## CONTENTS

About This Chapter .....	11-3
Defining New Data Types and New Names for Existing Data Types .....	11-4
Arrays .....	11-8
Enumerations .....	11-17
Defining an Enumeration Data Type .....	11-17
Defining a Variable that has an Enumeration Type .....	11-19
Defining an Enumeration Type and Enumeration Variables in the Same Statement .....	11-19
Pointers .....	11-22
Using Pointers .....	11-24
Pointer Arithmetic .....	11-25
Passing Pointers to Functions .....	11-26
Structures .....	11-30
Defining a Structure Data Type .....	11-31
Defining a Variable that has a Structure Data Type .....	11-32
Defining a Structure Type and Structure Variables in the Same Statement .....	11-34
Defining and Using Bit Fields .....	11-35
Unions .....	11-39
Defining a Union Data Type .....	11-39
Defining a Variable that has a Union Data Type .....	11-40
Defining a Union Type and a Union Variable in the Same Statement .....	11-40

---

## About This Chapter

This chapter describes how to define and declare variables having the following complex data types:

- Arrays
- Enumerations
- Pointers
- Structures
- Unions.

This chapter also explains how to use the **typedef** keyword to define synonyms for data types.

Chapter 10, “Primary Data Types” describes the following primary data types:

- Characters
- Integers
- Floating-point numbers
- **void** functions.

---

# Defining New Data Types and New Names for Existing Data Types

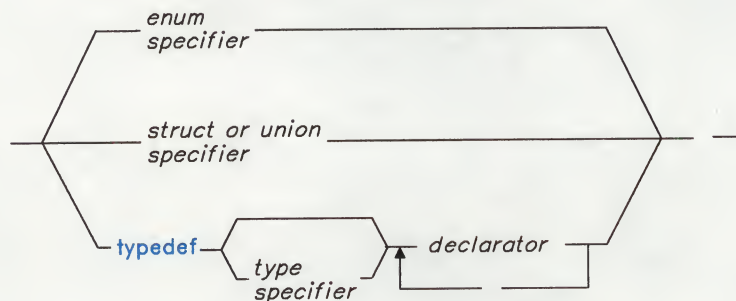
---

## Description

The C language provides you with several ways for defining new data types. You can define structures, unions, or enumerated data types. Also, the **typedef** keyword enables you to define a synonym for a data type.

**Type definitions** must appear outside a function or at the beginning of a block. A type definition has the form:

*type  
definition*



See “Enumerations” on page 11-17, “Structures” on page 11-30, and “Unions” on page 11-39 for more information about defining enumeration, structure, and union data types. The following paragraphs describe how to define synonyms for data types.

The **typedef** keyword enables you to define a synonym for a data type. The synonym does not become a new data type. It becomes a new name for an existing data type. A type definition does not reserve any storage.

A synonym definition contains the **typedef** keyword followed by a type specifier (optional) and a comma-separated list of declarators. The type specifier can be a primary data type keyword, such as **int**, or the name of a program-defined data type, such as the name of a union type. If the **typedef** definition does not contain a type specifier, the definition creates synonyms for the keyword **int**. The declarator can be an identifier, which becomes the synonym, or some combination of symbols, the keyword **volatile**, and an identifier. The variables defined using the synonym as the data type automatically receive the attributes listed in the type definition declarator. For example:

---

```
typedef short volatile *outer;  
outer truval = 0;
```

The **typedef** defines `outer` as a synonym for the type pointer to a **volatile short**. This definition is followed by a definition of the data object `truval`. `truval` becomes a pointer to a **volatile short** that has the initial value 0 (zero).

The following example defines `Code` as a synonym for `int`:

```
typedef int Code;
```

Since `Code` and `int` name the same data type, the following two definitions are equivalent:

```
Code patient, doctor, procedure;  
int patient, doctor, procedure;
```

A **typedef** name can be used in a cast (see “Cast” on page 13-20). The following example defines `Whole` as a synonym for `int` and converts the floating-point numbers 123.45 and 567.89 into `Whole` (or **int**) numbers:

```
typedef int Whole;  
(Whole) 123.45 + (Whole) 567.89;
```

The following expression is the result of the previous cast:

```
123 + 567
```

You can use **typedef** names to increase the portability of a program. In a program that contains integer variables that require different sizes of storage, you can define a synonym for each storage size (**short**, **int**, and **long**). When you port the program to various systems, you can change the **typedef** definitions to specify the appropriate sizes on that system. Changing three **typedef** definitions can take far less time than changing numerous variable definitions in a large program.

Improving program readability is another reason for defining synonyms. The following definition clarifies that `i` and `j` are used for subscripting purposes:

```
typedef int Subscript;  
Subscript i, j;
```

For simple cases, you can achieve the same results by using the **define** preprocessor statement. For example:

```
#define Subscript int  
Subscript i, j;
```

The preprocessor reads the **define** statement, while the compiler reads the **typedef** definition. See “**define**” on page 16-5 for more information about the **define** preprocessor statement.

---

The **typedef** definition enables you to create synonyms for complex data types. For example:

```
typedef char *Array_ptr;
```

defines `Array_ptr` as a synonym for "a pointer to a character." The `Array_ptr typedef` name can be used as follows:

```
Array_ptr search(line, letter)
char line[ ];
char letter;
{
    Array_ptr pline = line;

    *pline = letter;
    return(pline);
}
```

The following example defines `asset` and `liability` as synonyms for the type `struct account`:

```
typedef struct account
{
    int number;
    char description[15];
    float balance;
} asset, liability;
```

Because the keyword **struct** followed by the tag `account` is interchangeable with the **typedef** names `asset` and `liabilities`, the data definition:

```
struct account receivables, payables;
```

is equivalent to the following two data definitions:

```
asset receivables;
liability payables;
```

---

Although you can define **typedef** names for **struct** and **union** tags, you cannot use a **typedef** name in place of a tag in a self-referential structure or union. In the following example, you cannot replace `struct account` in line 4 with `bank_account`:

```
1  typedef struct account
2  {
3      int number;
4      struct account *next;
5  } bank_account;
```

## Related Information

“Defining an Enumeration Data Type” on page 11-17.

“Defining an Enumeration Type and Enumeration Variables in the Same Statement” on page 11-19.

“Defining a Structure Data Type” on page 11-31.

“Defining a Structure Type and Structure Variables in the Same Statement” on page 11-34.

“Defining a Union Data Type” on page 11-39.

“Defining a Union Type and a Union Variable in the Same Statement” on page 11-40.

“Declarators” on page 8-8.

---

# Arrays

---

## Description

An **array** contains an ordered group of data objects. Each object is called an **element**. All elements within an array have the same data type.

## Storage

You can define arrays as having any storage class. Most systems, however, treat arrays defined with the **register** storage class as having the **auto** storage class.

## Type Specifier

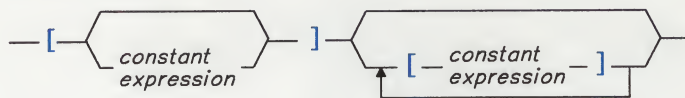
You can use any type specifier in an array definition or declaration. Thus, arrays can be of any data type, except function. (You can, however, declare an array of pointers to functions.)

## Declarator

The declarator contains an identifier followed by a **subscript declarator**. The identifier can be preceded by an \* (asterisk), making the variable an array of pointers.

The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. A subscript declarator has the form:

*subscript  
declarator*



Each bracketed expression describes a different dimension. If the brackets contain a constant expression, the constant expression must have an integral value. The IBM RT PC C Language compiler converts this value to type **int**. The value of the constant expression determines the number of elements in that dimension. The following example defines a one-dimensional array that contains four elements having type **char**:

```
char list[4];
```

---

The first subscript of each dimension is 0 (zero). Thus, the array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

The following example defines a two-dimensional array that contains six elements of type `int`:

```
int roster[3][2];
```

In multi-dimensional arrays, when referencing elements in order of increasing storage location, the last subscript varies the fastest. Thus, the array `roster` contains the elements:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

You can leave the first (and only the first) set of subscript brackets empty in:

- External definitions that contain initializations
- **static** definitions that contain initializations
- **extern** declarations
- Parameter declarations (see “Using Arrays” on page 11-13).

In **extern** and **static** array definitions that leave the first set of subscript brackets empty, the compiler uses the initializer to determine the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multi-dimensional array, the compiler compares the initializer to the subscript declarator in order to determine the number of elements in the first dimension.

In an **extern** declaration, the compiler refers to the associated external definition to determine the number of elements in the first dimension.

The C language does not limit the number of dimensions in an array or of elements in a dimension. Some compilers, and all systems, impose limits. The IBM RT PC C Language compiler imposes no limits on the number of dimensions in an array or the number of elements in a dimension.

---

## Initializer

You can initialize arrays in **extern** and **static** definitions, only. The initializer contains the symbol = followed by a brace-enclosed comma-separated list of constant expressions. You do not need to initialize all elements in an array. Elements that are not initialized (in **extern** and **static** definitions, only) receive the value 0 (zero).

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values:

Element	Value
---------	-------

<code>number[0]</code>	5
<code>number[1]</code>	7
<code>number[2]</code>	2

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:

Element	Value
---------	-------

<code>number1[0]</code>	5
<code>number1[1]</code>	7
<code>number1[2]</code>	0

Instead of using an expression in the subscript declarator to define the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements:

Element	Value
---------	-------

<code>item[0]</code>	1
<code>item[1]</code>	2
<code>item[2]</code>	3
<code>item[3]</code>	4
<code>item[4]</code>	5

---

You can initialize a one-dimensional character array by specifying:

- A brace enclosed comma-separated list of character constants
- A string constant (braces surrounding the constant are optional).

If you specify a string constant, the compiler places the character `\0` (NUL) at the end of the string.

The following definitions show character array initializations:

```
static char name1[ ] = { 'J', 'a', 'n' };  
static char name2[ ] = { "Jan" };
```

These two definitions create the following elements:

Element	Value	Element	Value
name1[0]	J	name2[0]	J
name1[1]	a	name2[1]	a
name1[2]	n	name2[2]	n
		name2[3]	\0

You can initialize a multi-dimensional array by:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by incrementing the subscript of the last dimension fastest. This form of a multi-dimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

```
static month_days[2][12] =  
{  
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,  
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31  
};
```

- Using braces to group the values of the elements you want initialized. You can place braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension. (You can consider these elements as rows.) The initialization contains braces around each of these two elements:

```
static int month_days[2][12] =  
{  
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },  
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }  
};
```

You can use nested braces to selectively initialize dimensions and elements in a dimension. The braces do not change the order in which the compiler assigns values. The following definition explicitly initializes six elements in a 12-element array:

```
static matrix[3][4] =  
{  
    {1, 1},  
    {1, 1},  
    {1, 1}  
};
```

The initial values of `matrix` are:

Element	Value	Element	Value
<code>matrix[0][0]</code>	1	<code>matrix[1][2]</code>	0
<code>matrix[0][1]</code>	1	<code>matrix[1][3]</code>	0
<code>matrix[0][2]</code>	0	<code>matrix[2][0]</code>	1
<code>matrix[0][3]</code>	0	<code>matrix[2][1]</code>	1
<code>matrix[1][0]</code>	1	<code>matrix[2][2]</code>	0
<code>matrix[1][1]</code>	1	<code>matrix[2][3]</code>	0

---

## Using Arrays

An unadorned array name (for example, `region` instead of `region[4]`) is a pointer to the beginning of the array. When you pass an array as a parameter, the called function receives the address of the array. Through this pointer, the called function can change the value of any element in the array.

The following example shows a parameter declaration for a one-dimensional array:

```
test(y)
int y[ ];
{
    .
    .
    .
}
```

The following example shows a parameter declaration for a two-dimensional array:

```
sum(x)
int (*x)[5];
{
    .
    .
    .
}
```

The parameter declaration:

```
int (*x)[5];
```

declares a pointer to an array of five `int` values. This declaration could be written in the following way, also:

```
int x[ ][5];
```

---

## Examples

The following program defines two floating-point arrays: `prices` and `total`. Only `prices` is initialized.

The first **for** statement prints the values of the elements in `prices`. The second **for** statement adds five percent to the value of each element in `prices`, assigns the result to an element in `total`, and prints the value of each element in `total`.

```
/* Example of one-dimensional arrays. */

#include <stdio.h>

main()
{
    static float prices[ ] = { 1.41, 1.50, 3.75, 5.00, .86 };
    auto float total[5];
    int i;

    for (i = 0; i < 5; i++)
    {
        printf("price = %.2f\n", prices[i]);
    }

    printf("\n\n");

    for (i = 0; i < 5; i++)
    {
        total[i] = prices[i] * 1.05;

        printf("total = %.2f\n", total[i]);
    }
}
```

---

The preceding program produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86
```

```
total = $1.48
total = $1.58
total = $3.94
total = $5.25
total = $0.90
```

The following program defines the multi-dimensional array `salary_tbl`. A `for` loop prints the values of `salary_tbl`.

```
/* example of a multi-dimensional array */

#include <stdio.h>

main()
{
    static int salary_tbl[3][5] =
    {
        { 500, 550, 600, 650, 700 },
        { 600, 670, 740, 810, 880 },
        { 740, 840, 940, 1040, 1140 }
    };
    int grade , step;

    for (grade = 0; grade < 3; grade++)
        for (step = 0; step < 5; step++)
        {
            printf("salary_tbl[%d] [%d] = %d\n", grade, step,
                    salary_tbl[grade] [step]);
        }
}
```

---

The preceding program produces the following output:

```
salary_tbl[0][0] = 500
salary_tbl[0][1] = 550
salary_tbl[0][2] = 600
salary_tbl[0][3] = 650
salary_tbl[0][4] = 700
salary_tbl[1][0] = 600
salary_tbl[1][1] = 670
salary_tbl[1][2] = 740
salary_tbl[1][3] = 810
salary_tbl[1][4] = 880
salary_tbl[2][0] = 740
salary_tbl[2][1] = 840
salary_tbl[2][2] = 940
salary_tbl[2][3] = 1040
salary_tbl[2][4] = 1140
```

## Related Information

“Pointers” on page 11-22.  
“Array Subscripts” on page 13-14.  
“String” on page 9-21.  
“Declarators” on page 8-8.  
“Initializers” on page 8-12.  
Chapter 14, “Conversions.”

---

# Enumerations

---

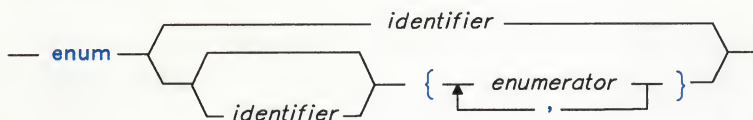
## Description

An **enumeration** data type represents a set of values that you define. You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can separate the definition of the enumeration data type from all variable definitions.

## Defining an Enumeration Data Type

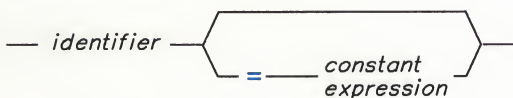
An enumeration type definition contains the **enum** keyword followed by an identifier (the enumeration tag) and a brace-enclosed list of enumerators. Each enumerator is separated by a comma. The shaded area of the following diagram shows the form of an enumeration type definition:

*enum  
specifier*



The identifier names the data type (just as **int** names the integer data type). The list of enumerators provides the data type with a set of values. Each **enumerator** represents an **integer** value. An enumerator has the form:

*enumerator*



---

The identifier in an enumerator is called an *enumeration constant*. You can use an enumeration constant anywhere an integer constant is allowed. The compiler determines the integer representation of an enumerator (and its corresponding enumeration constant) by the following rules:

1. If an = (equal sign) and a constant expression follow the identifier, the identifier represents the value of the constant expression.
2. Otherwise, if the enumerator is the leftmost value in the list, the identifier represents the value 0 (zero).
3. Otherwise, the identifier represents the integer value that is one greater than the value represented by the preceding enumerator.

The following example defines the enumeration data type status:

```
enum status { run, create, delete=5, suspend };
```

The data type status represents the following values only:

Enumeration Constant	Integer Representation
run	0
create	1
delete	5
suspend	6

Each enumeration constant must be unique within the block or the file where the enumeration data type is defined. In the following example, the re-definition of average on line 4 and of poor on line 5 cause compiler error messages:

```
1 func()  
2 {  
3     enum score { poor, average, good };  
4     enum rating { below, average, above };  
5     int poor;  
6 }
```

---

## Defining a Variable that has an Enumeration Type

An enumeration variable definition is similar to any other type of variable definition. It contains a storage class specifier (optional), a type specifier, a declarator, and an initializer (optional). The type specifier, however, has a special form. The type specifier contains the keyword **enum** followed by the name of the enumeration data type. You must define the enumeration data type before you can define a variable having that type.

The first line of the following example defines the enumeration data type `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (2). The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`:

```
enum grain { oats, wheat, barley, corn, rice };  
enum grain g_food = barley;
```

The initializer for an enumeration variable contains the symbol `=` followed by an expression. The expression must evaluate to an **int** value.

## Defining an Enumeration Type and Enumeration Variables in the Same Statement

You can place a type definition and a variable definition in one statement by placing a declarator and an optional initializer after the type definition. If you want to specify a storage class for the variable, you must place the storage class specifier at the beginning of the statement. For example:

```
register enum score { poor=1, average, good } rating = good;
```

The previous example is equivalent to the following two statements:

```
enum score { poor=1, average, good };  
register enum score rating = good;
```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class `register`, the data type `enum score`, and the initial value 3 (or `good`).

If you combine a data type definition with the definitions of all variables having that data type, you can leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }  
    weekday;
```

This example defines the variable `weekday`. `weekday` can be assigned any of the specified enumeration constants, as well as any **int** value.

---

If you place a type definition in the same statement as a definition of a variable having the **volatile** attribute, the **volatile** attribute applies to that variable only. For example:

```
enum shape { round, square, triangular, oblong } volatile object;
enum shape appearance;
```

The variable `object` is defined as having the **volatile** attribute. The variable `appearance` does not receive the **volatile** attribute.

## Examples

The following program receives an integer as input. The output is the German name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints `Keintag`.

```
#include <stdio.h>

enum days {
    Monday=1, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
} weekday;

main()
{
    printf("Enter an integer for the day of the week.\n");
    scanf("%d", &weekday);
    german(weekday);
}

german(weekday)
enum days weekday;
{
    switch (weekday)
    {
        case Monday:
            printf("Montag\n");
            break;
```

---

```
    case Tuesday:
        printf("Dienstag\n");
        break;

    case Wednesday:
        printf("Mittwoch\n");
        break;

    case Thursday:
        printf("Donnerstag\n");
        break;

    case Friday:
        printf("Freitag\n");
        break;

    case Saturday:
        printf("Samstag\n");
        break;

    case Sunday:
        printf("Sonntag\n");
        break;

    default:
        printf("Keintag\n");
}
}
```

## Related Information

“Enumeration” on page 9-8.  
“Constant Expression” on page 13-9.  
“Identifiers” on page 7-6.

---

# Pointers

---

## Description

A **pointer** type variable holds the address of a data object. A pointer can refer to an object of any one data type, but never to an object having **register** storage class. Some common uses for pointers are:

- To pass the address of a variable to a function. By referencing the address of a variable, a function can change the contents of that variable. See “Calling Functions and Passing Values” on page 12-12.
- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure.
- To treat an array of type **char** as a string.

## Storage Class

A pointer can have any storage class.

## Type Specifier

You can use any type specifier in a pointer definition or declaration. A pointer should have the same type specifier as the object to which it points.

## Declarator

The symbol **\*** precedes the identifier. If the keyword **volatile** appears before the **\***, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** comes between the **\*** and the identifier, the declarator describes a **volatile** pointer.

The following example declares `pcoat` as a pointer to an object having type **long**:

```
extern long *pcoat;
```

---

The following example declares `pnut` as a pointer to an **int** object having the **volatile** attribute:

```
extern int volatile *pnut;
```

The following example defines (and declares) `psoup` as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines (and declares) `pfowl` as a pointer to an enumeration object of type `bird`:

```
enum bird *pfowl;
```

The next example declares `x` as a function that returns an object having type pointer to a **char**:

```
char *x();
```

## Initializer

The usual initializer is an `=` (equal sign) followed by the address that the pointer is to contain. The following example defines the variables `time` and `speed` as having type **double** and `amount` as having type pointer to a **double**. `amount` is initialized to point to `time`:

```
double time, speed, *amount = &time;
```

The compiler converts the name of an array (not containing a subscript) to a pointer to the first element in the array. Thus, you assign the address of the first element of an array to a pointer by specifying the name of the array. If `class` is an array, the following two definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in `class`:

```
int *student = class;  
int *student = &class[0];
```

---

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer. The following example defines the pointer variable `string` and the string constant `"abcd"`. `string` is initialized to point to `a`.

```
char *string = "abcd";
```

The following example defines `weekdays` as an array of pointers to string constants. Each element points to a different string. The object `weekdays[2]`, for example, points to the string `"Tuesday"`.

```
static char *weekdays[ ] =  
    {  
        "Sunday", "Monday", "Tuesday", "Wednesday",  
        "Thursday", "Friday", "Saturday"  
    };
```

A pointer can also be initialized to the integer constant 0 (zero). Such a pointer is called a *null pointer*. A null pointer does not point to any object.

## Using Pointers

Two operators are commonly used in accessing pointers, the `&` (address) operator and the `*` (indirection) operator. You can use the `&` operator to reference the address of an object. For example, the following statement assigns the address of `x` to the variable `p_to_x`. The variable `p_to_x` has been defined as a pointer.

```
p_to_x = &x;
```

The `*` (indirection) operator enables you to access the value of the object to which a pointer refers. The following statement assigns to `y`, the value of the object to which `p_to_x` points:

```
y = *p_to_x;
```

The following statement assigns the value of `y` to the variable that `*p_to_x` references:

```
*p_to_x = y;
```

---

## Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and Decrement
- Addition and Subtraction
- Comparison
- Assignment.

The ++ (increment) operator increases the value of a pointer by the size of the data object to which the pointer refers. If the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The -- (decrement) operator decreases the value of a pointer by the size of the data object to which the pointer refers. If the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add an integer to or subtract an integer from a pointer. The compiler multiplies the integer by the size of the data object to which the pointer refers. Then the compiler adds (or subtracts, as indicated by the operator) this value to the value of the pointer. If the pointer `p` points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

You can subtract one pointer from another pointer. This operation yields the number of elements in the array that separate the two address to which the pointers refer.

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`.

You can assign to a pointer the address of a data object, the value of another pointer, or the constant 0 (zero).

---

## Passing Pointers to Functions

Pointers provide an indirect way for a called function to alter the value of a variable in the calling function. A called function receives only a copy and cannot alter the value of the pointer in the calling function. However, the called function can alter the value of the variable that the pointer points to. The following program shows how you can pass a pointer to a function and change the value of the object to which the pointer points:

```
/*
** This program accepts a time value for a timer
** and then times down the timer.
*/

#include <stdio.h>

main()
{
    int t_timer;

    printf("Set timer to:\n");
    scanf("%d", &t_timer);
    while (! (count_down(&t_timer) ) )    /* while not timed down */
    {
        printf("Timer still timing. %d\n", t_timer);
    }
    printf("Timer has timed down.\n");
}    /* End main    */
```

---

```
/*  
** This function decrements a timer and returns  
** true if the timer times down to zero and returns  
** false if the timer is already zero or not timed  
** down.  
*/
```

```
int count_down(timer)  
int *timer;  
{  
    return(*timer ? !--(*timer) : *timer);  
} /* End count_down */
```

Interaction with the preceding program could produce the following session:

```
Output:  Set timer to:  
Input:   6  
Output:  Timer still timing. 5  
         Timer still timing. 4  
         Timer still timing. 3  
         Timer still timing. 2  
         Timer still timing. 1  
         Timer has timed down.
```

---

## Examples

The following program contains pointer arrays:

```
/*
** Program to search for the first occurrence
** of a specified character string
*/

#include <stdio.h>

main()
{
    static char *names[ ] = { "Jim", "Amy", "Mark", "Alice", NULL };
    char new_nm[ ], *nm_pointer, *find_name();

    printf("Enter name to be searched.\n");
    scanf("%s", new_nm);
    printf("name %s found\n",
           ((nm_pointer = find_name(names, new_nm)) == NULL)
            ? "not" : nm_pointer);
} /* End of main */

/* Function to find a specified name */

char *find_name(array, strng)
char **array; /* pointer to arrays of pointers */
char *strng; /* pointer to character array entered */
{
    for ( ; *array != NULL; array++) /* for each name */
    {
        if (strcmp(*array, strng) == 0) /* if strings match */
            return(*array); /* return pointer to name */
    }
    return(NULL); /* name not found */
} /* End of find_name */
```

---

Interaction with the preceding program could produce the following session:

Output: Enter name to be searched.

Input: Mark

Output: name Mark found

## Related Information

“Address” on page 13-19.

“Indirection” on page 13-20.

“Declarators” on page 8-8.

“Initializers” on page 8-12.

---

# Structures

---

## Description

A **structure** contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is called a **member** or **field**.

You can use structures to define several sets of similar variables. For example, if you want to allocate storage for the components of an address, you can define the following variables:

```
char *street;
char city[MAX_CITY];
char state[MAX_STATE];
int zip_code;
```

However, if you want to allocate storage for more than one address, you can group the components of each address by defining a structure data type and several variables having the structure data type:

```
1 struct address {
2     char *street;
3     char city[MAX_CITY];
4     char state[MAX_STATE];
5     int zip_code;
6 };
7 struct address perm_address;
8 struct address temp_address;
```

Lines 1 through 6 define the structure data type `address`. Line 7 defines the variable `perm_address`, and line 8 defines the variable `temp_address`, both of which are instances of the structure `address`. Both `perm_address` and `temp_address` contain the members described in lines 2 through 5.

You can reference a member of a structure by specifying the structure variable name, the `.` (dot operator), and the member name. For example:

```
perm_address.state[0] = 'A'
```

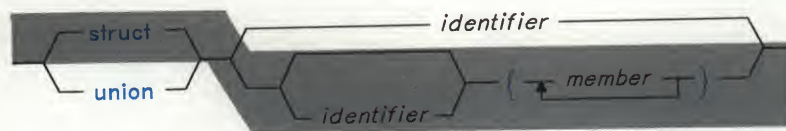
assigns the value 'A' to the first element of the array `state` that is in the structure `perm_address`.

## Defining a Structure Data Type

A structure type definition does not allocate storage. It describes the members that are part of the structure.

A structure type definition contains the **struct** keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members. The shaded area of the following diagram shows the form of a structure type definition:

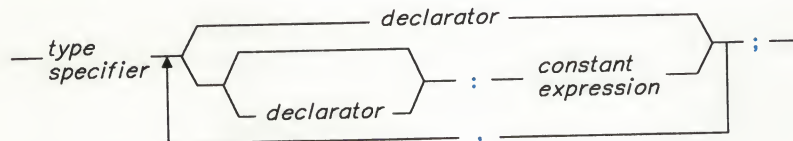
*struct or union  
specifier*



The identifier names the data type. If you do not name the data type, you must place all variable definitions that refer to that data type within the statement that defines the data type.

The list of members provides the data type with a description of the values that can be stored in the structure. A member has the form:

*member*



A member can be of any data type and can have the **volatile** attribute. Additionally, if a : (colon) and a constant expression follow the declarator, the member represents a **bit field**.

The names of structure types and members do not conflict with other identifiers. You cannot use the name of a member more than once in a structure type. However, you can use the same member name in another structure type that is defined within the same scope.

You cannot define a structure type that contains itself as a member. You can, however, define a structure type that contains a pointer to itself as a member.

---

## Defining a Variable that has a Structure Data Type

A structure variable definition contains a storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

### Storage

You can define structures having any storage class. Most systems, however, treat structures defined with the register storage class as automatic structures.

### Type Specifier

The type specifier contains the keyword **struct** followed by the name of the structure data type. You must define the structure data type before you can define a structure having that type.

### Declarator

The declarator for a structure variable is an identifier. If the symbol **\*** precedes the identifier, the identifier names a pointer to a structure of the specified data type. If a constant expression enclosed in `[ ]` (brackets) follows the identifier, the identifier names an array of structures of the specified data type. If **\*** precedes the identifier and a constant expression enclosed in `[ ]` follows the identifier, the identifier names an array of pointers to structures of the specified data type.

### Initializer

You can initialize structures in external and static definitions, only. The initializer contains an `=` (equal sign) followed by a brace-enclosed comma-separated list of values. You do not need to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {  
    char *street;  
    char city[MAX_CITY];  
    char state[MAX_STATE];  
    int zip_code;  
};  
static struct address perm_address =  
    { "1602 Maple St.", "Austin", "Texas", 78753 };
```

---

The values of perm\_address are:

Member	Value
*perm_address.street	"1602 Maple St."
perm_address.city	"Austin"
perm_address.state	"Texas"
perm_address.zip_code	78753

The following definition shows a partially initialized structure:

```
struct address {  
    char *street;  
    char city[MAX_CITY];  
    char state[MAX_STATE];  
    int zip_code;  
};  
struct address temp_address =  
    { "2206 Washington St.", "Omaha", "Nebraska" };
```

The values of temp\_address are:

Member	Value
*temp_address.street	"2206 Washington St."
temp_address.city	"Omaha"
temp_address.state	"Nebraska"
temp_address.zip_code	0

---

## Defining a Structure Type and Structure Variables in the Same Statement

You can place a type definition and a variable definition in one statement by placing a declarator and an initializer (optional) after the type definition. If you want to specify a storage class for the variable, you must place the storage class specifier at the beginning of the statement. For example:

```
static struct {  
    char *street;  
    char city[MAX_CITY];  
    char state[MAX_STATE];  
    int zip-code;  
} perm-address, temp-address;
```

The preceding example does not name the structure data type. Thus, `perm-address` and `temp-address` are the only structure variables that will be assigned this data type. If an identifier were placed after `struct`, additional variable definitions of this data type could be made later in the program.

The structure type (or tag) cannot have the **volatile** attribute, but a member or a structure variable can be defined as having the **volatile** attribute. For example:

```
static struct class1 {  
    char [20] descript;  
    volatile long code;  
    short complete;  
} volatile file1, file2;  
struct class1 subfile;
```

This example gives the **volatile** attribute to the structure variable `file1` and to the structure members `file2.code` and `subfile.code`.

---

## Defining and Using Bit Fields

A structure can contain packed data known as **bit fields**. You can use bit fields for data that requires just a few bits of storage. A bit field definition contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon. The constant expression specifies how many bits the field reserves. A bit field that is defined as having a length of 0 (zero) causes the next field to be aligned on the next word boundary. A bit field that is larger than the remaining space in the current word is automatically placed in the next word. You can initialize a bit field.

The maximum size of a field is a word. The IBM RT PC C Language compiler has a 32-bit wordsize and assigns bit fields from left to right. You cannot define an array of bit fields, and you cannot take the address of a bit field.

You can define a bit field as having type **int** or **unsigned int**. The IBM RT PC C Language compiler, however, always takes the type of a bit field to be **unsigned int**.

The following example defines the structure type switches and the structure kitchen, which has the type switches:

```
struct switches {  
    unsigned light : 1;  
    unsigned toaster : 1;  
    int count;  
    unsigned ac : 4;  
    unsigned : 4;  
    unsigned clock : 1;  
    unsigned : 0;  
    unsigned flag : 1;  
} kitchen ;
```

The structure `kitchen` contains six members. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
<code>light</code>	1 bit
<code>toaster</code>	1 bit
<code>count</code>	The size of an <code>int</code>
<code>ac</code>	4 bits
	4 bits (unnamed field)
<code>clock</code>	1 bit
	undefined number of bits (unnamed field)
<code>flag</code>	1 bit

The fields `light` and `toaster` each require 1 bit of storage. These members are assigned storage next to each other in the same word. `count` is stored in the next word. `ac` requires 4 bits of storage and is aligned on the next word boundary. The next bit field has no name. This unnamed field uses 4 bits to separate `ac` and `clock`. `clock` is stored in the following bit. The unnamed field with a length of 0 (zero) forces `flag` to be on the next word boundary.

All references to structure fields must be fully qualified. Therefore, you cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.

The following expression sets the `light` field to 1:

```
kitchen.light = 1
```

The following expression sets the `toaster` field to 0 because only the first low-order bit is assigned to the `toaster` field:

```
kitchen.toaster = 2
```

---

## Examples

The following program finds the sum of the integer numbers in a linked list:

```
/* program to illustrate linked lists */
#include <stdio.h>

struct record {
    int number;
    struct record *next_num;
};

main()
{
    struct record name1, name2, name3;
    struct record *recd_pointer = &name1;
    int sum = 0;

    name1.number = 144;
    name2.number = 203;
    name3.number = 488;

    name1.next_num = &name2;
    name2.next_num = &name3;
    name3.next_num = NULL;

    while (recd_pointer != NULL)
    {
        sum += recd_pointer->number;
        recd_pointer = recd_pointer->next_num;
    }
    printf("Sum = %d\n", sum);
}
```

---

The structure type `record` contains two members: `number` (an integer) and `next_num` (a pointer to a structure variable of type `record`).

The record type variables `name1`, `name2`, and `name3` are assigned the following values:

Member Name	Value
<code>name1.number</code>	144
<code>name1.next_num</code>	The address of <code>name2</code>
<code>name2.number</code>	203
<code>name2.next_num</code>	The address of <code>name3</code>
<code>name3.number</code>	488
<code>name3.next_num</code>	NULL (Indicating the end of the linked list.)

The variable `recd_pointer` is a pointer to a structure of type `record`. `recd_pointer` is initialized to the address of `name1` (the beginning of the linked list).

The while loop causes the linked list to be sequenced through until `recd_pointer` equals NULL. The statement:

```
recd_pointer = recd_pointer->next_num;
```

advances the pointer to the next object in the list.

## Related Information

“Structure and Union Member Specifications” on page 13-15.

“Declarators” on page 8-8.

“Initializers” on page 8-12.

---

# Unions

---

## Description

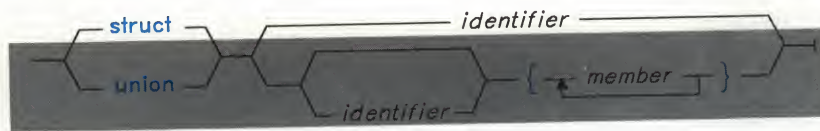
A **union** is an object that can hold any one of several data types.

## Defining a Union Data Type

A union type definition does not allocate storage.

A union type definition contains the **union** keyword followed by an identifier (optional) and a brace-enclosed list of members. The shaded area of the following diagram shows the form of a union type definition:

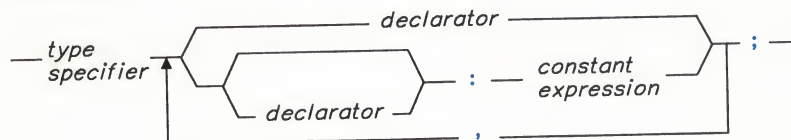
*struct or union  
specifier*



The identifier names the data type. If you do not name the data type, you must place all variable definitions that refer to that data type within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union. A member has the form:

*member*



---

You can reference one of the possible members of a union like you reference a member of a structure. For example:

```
address.street[0] = '\n';
```

assigns '\n' to the first element in the character array `street`, a member of the union `address` (assuming `address` was defined as a union).

## Defining a Variable that has a Union Data Type

A union variable definition contains a storage class keyword, the **union** keyword, a union tag, and a declarator. The union tag indicates the data type of the union variable.

### Type Specifier

The type specifier contains the keyword **union** followed by the name of the union data type. You must define the union data type before you can define a union having that type.

You can define a union data type and a union of that type in the same line of code by placing the variable declarator after the data type definition.

### Declarator

The declarator is an identifier, possibly with the **volatile** attribute.

### Initializer

You cannot initialize a union.

## Defining a Union Type and a Union Variable in the Same Statement

You can place a type definition and a variable definition in one statement by placing a declarator after the type definition. If you want to specify a storage class for the variable, you must place the storage class specifier at the beginning of the statement.

Although the union type cannot be defined as having the **volatile** attribute, a member or a union variable can be defined as having the **volatile** attribute. For example:

```
union climate {  
    float temp;  
    volatile struct wind;  
    long code;  
} volatile current;  
union climate forecast;
```

---

This example gives the **volatile** attribute to the **union** variable `current` and to the union member `forecast.wind`.

## Examples

The following example defines a union data type (not named) and a union variable (named `piece_parts`). The member of `piece_parts` can be a **long int**, a **float**, or an array of type **char**.

```
union {
    long item_num;
    float price;
    char description[30];
} piece_parts;
```

The following example defines the union type `data` as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two data type variables: `input` and `output`.

```
union data {
    char charctr;
    int whole;
    float real;
};
union data input, output;
```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a real number to `output`:

```
output.real = 9 / 3;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the **union** variable `input`. `input` has the **union** data type defined in the previous example.

```
struct {
    int id_num;
    int type_of_input;
    union data input;
} records[10];
```

---

The following statement assigns a character to the structure member `input` of the first element of records:

```
records[0].input.charctr = '\1';
```

## Related Information

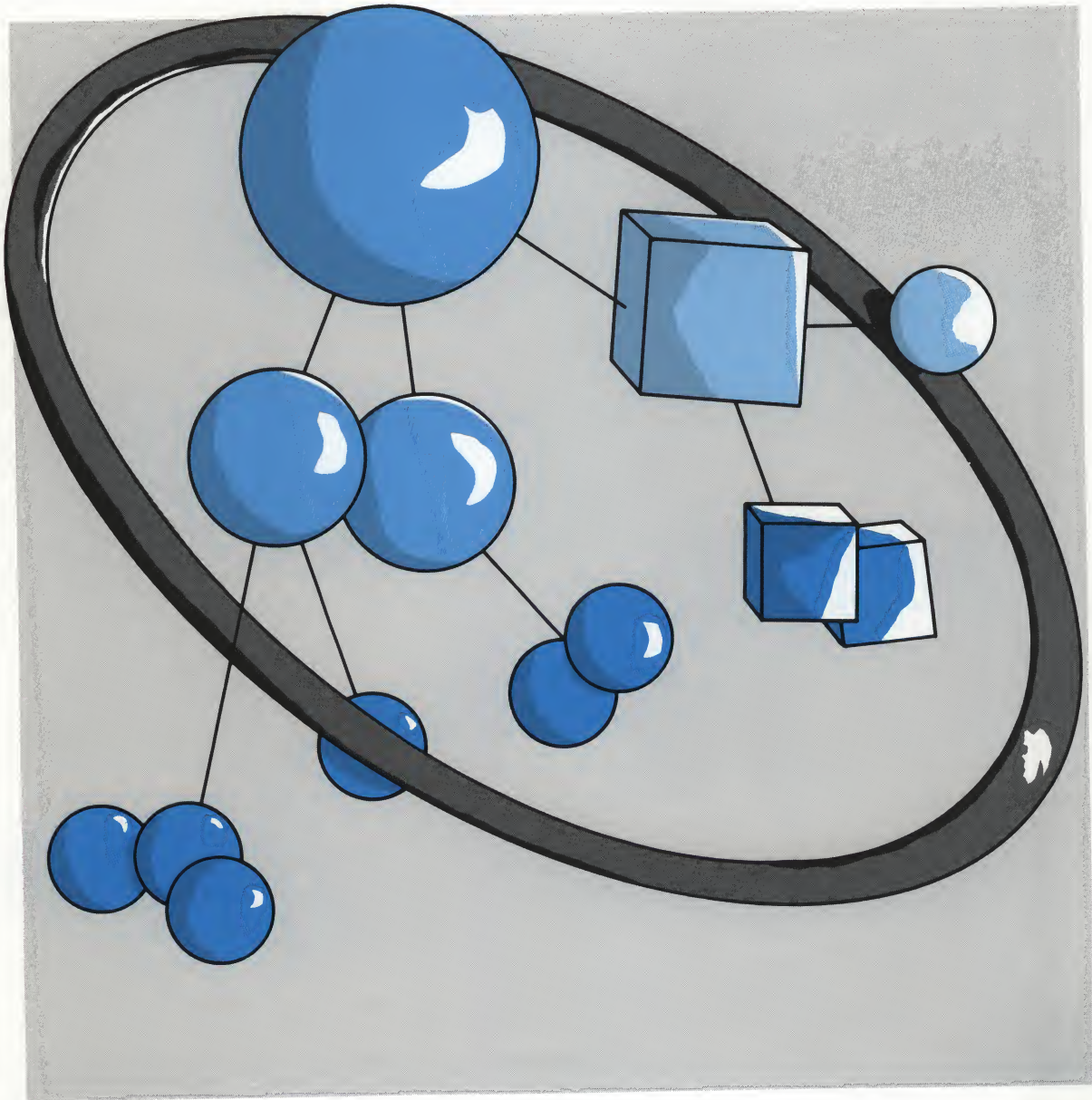
“Structure and Union Member Specifications” on page 13-15.

“Declarators” on page 8-8.

“Initializers” on page 8-12.

---

## Chapter 12. Functions



---

## CONTENTS

About This Chapter .....	12-3
main .....	12-4
Function Definition .....	12-5
Function Declarator .....	12-6
Parameter Declaration .....	12-7
Function Body .....	12-9
Function Declarations .....	12-10
Calling Functions and Passing Values .....	12-12

---

## About This Chapter

This chapter describes the structure and usage of functions.

---

# main

---

## Description

When you begin the execution of a program the system automatically calls the function **main**. Therefore, every program must have exactly one function named **main**. A **main** function has the form:

*main*  
function



The function **main** can declare zero to three parameters. The first parameter, **argc**, has type **int** and indicates how many arguments were entered on the command line. The second parameter, **argv**, has type array of pointers to **char** objects. The value of **argc** indicates the number of pointers in the array **argv**. The first element in **argv** always points to a character array that contains the name (as invoked) of the program that is executing. The third parameter, **envp**, has type array of pointers to **char** objects. The array **envp** contains pointers to the environment of the program. The system determines the value of this parameter during program initialization (before calling **main**). You can access the value of this pointer using the function **getenv**. Thus, there is no need to declare the third parameter. Some operating systems do not generate **envp** parameters. The AIX Operating System, however, does generate **envp** parameters.

## Related Information

- “Function Definition” on page 12-5.
- “Calling Functions and Passing Values” on page 12-12.
- “Parameter Declaration” on page 12-7.
- Chapter 10, “Primary Data Types.”
- “Identifiers” on page 7-6.
- “Block” on page 15-4.

---

# Function Definition

---

## Description

A **function definition** describes a function. A function definition has the form:

*function  
definition*



A function definition contains the following:

1. The optional storage class specifier **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has the storage class **extern**.
2. An optional type specifier, which determines the type of value that the function returns. A function can have any type specifier. If a type specifier is not given, the function has the type specifier **int**.
3. A function declarator, which provides the function with a name, optionally further describes the type of the value that the function returns, and lists any parameters that the function expects. The declarator cannot describe a return type of function or array or any type having the volatile attribute.
4. Optional parameter declarations, which describe the values that the function receives. Parameters that are not declared have type **int**.
5. A block statement, which contains data definitions and code.

A function can be called by itself or by any function that appears in the same file as the function definition. If a function has the storage class **extern**, the function also can be called by functions that appear in other files. If a function has storage class **extern** and a return type other than **int**, the function definition or a declaration for the function must appear prior to, and in the same file as, a call to the function.

You cannot define an array of functions. You can, however, define an array of pointers to functions.

The following example is a complete definition of the function `sum`:

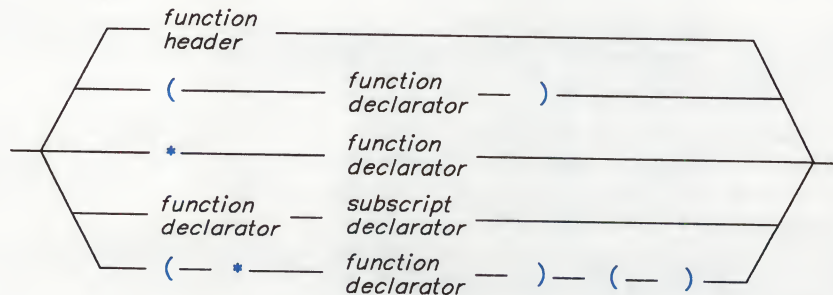
```
int sum(x, y)
int x, y;
{
    return(x + y);
}
```

The function `sum` has the storage class **extern**, returns an object that has type **int**, and receives two values declared as `x` and `y`. The function body contains a single statement that returns the sum of `x` and `y`.

## Function Declarator

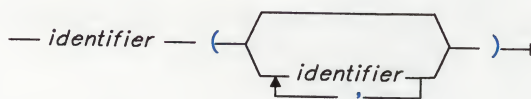
The **function declarator** names the function, provides additional information about the return value of the function, and lists the function parameters. A function declarator has the form:

*function  
declarator*



A function declarator must contain a **function header**. The function header names the function and lists the function parameters. The function header has the form:

*function  
header*



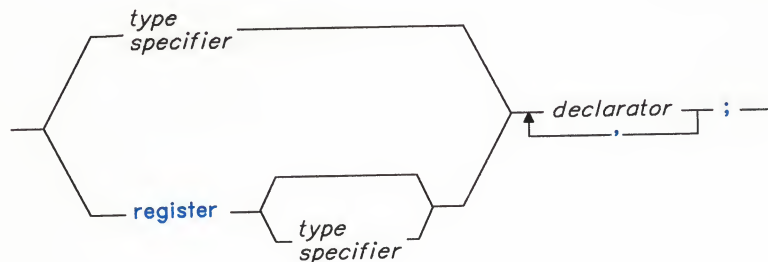
The following examples contain function declarators:

```
double square(x)
int area(x,y)
char *search(table)
static char *search()
char *search(days, weekday)
```

## Parameter Declaration

Each value that a function receives should be declared. This declaration is called a **parameter declaration**. A parameter declaration determines the storage class and the data type of the value. A parameter declaration has the form:

*parameter  
declaration*



You can request the **register** storage class and any type specifier for a parameter. If you do not specify the **register** storage class, the parameter receives the **auto** storage class. If you omit the type specifier, the compiler interprets the parameter as having type **int**.

If you do not provide a declaration for a parameter, the parameter receives the storage class **auto** and the data type **int**.

If a function does not receive any values, the function must not contain any parameter declarations. For example:

```
stop()
{
}
```

---

The following function `sort` contains two parameters, `table` and `length` in its parameter list. `table` is declared as a pointer to an array of integers. `length` is declared as an `int`. The local variables `i`, `j`, and `temp` must be defined within the function body.

```
sort(table, length)
int table[ ];
int length;
{
    int i, j, temp;

    for (i = 0; i < length - 1; i++)
        for (j = i + 1; j < length; j++)
            if (table[i] > table[j])
            {
                temp = table[i];
                table[i] = table[j];
                table[j] = temp;
            }
}
```

The following function `set_date` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class **register**.

```
set_date(date_ptr)
register struct date *date_ptr;
{
    date_ptr->mon = 2;
    date_ptr->day = 24;
    date_ptr->year = 85;
}
```

---

## Function Body

The body of a function is a block statement. The following function has an empty body:

```
void stub1()
{
}
```

The following function body contains a definition for the integer variable `big_num` and a call to the function `printf`:

```
largest(num1, num2)
int num1, num2;
{
    int big_num;

    if (num1 >= num2)
        big_num = num1;
    else
        big_num = num2;

    printf("big_num = %d\n", big_num);
}
```

---

# Function Declarations

---

## Description

A function is declared implicitly by its appearance in an expression. If the function has not been defined or declared previously, it is assumed to return a value having type **int**. A function cannot be declared as returning a data object having the **volatile** attribute.

If the called function is located after the function call and the called function returns a value that has a type other than **int**, you must declare the function prior to the function call.

## Examples

The following example defines the function `absolute` as having the return type `double`. Because this is a non-integer return type, `absolute` is declared above the function call.

```
#include <stdio.h>
main()
{
    double absolute();
    double f = 3;

    printf("absolute number = %f\n", absolute(f));
}

double absolute(number)
double number;
{
    if (number < 0)
        number = -number;

    return (number);
}
```

---

The following example defines the function `absolute` as having the return type `void`.  
The function `main` declares `absolute` as having the return type `void`.

```
main()
{
    void absolute();
    float f;

    absolute(f);
}

void absolute(number)
float number;
{
    if (number < 0)
        number = -number;

    printf("absolute number = %f\n" number);
}
```

## Related Information

“Declaration” on page 8-19.

“Scope of Functions” on page 8-20.

---

# Calling Functions and Passing Values

---

## Description

A function call specifies a function name and a list of arguments. The calling function passes the value of each argument to the specified function. The argument list is surrounded by parentheses, and each argument is separated by a comma. The argument list can be empty.

When an argument is passed in a function call, the function receives the value of the argument. If the value of the argument is an address, the called function can use indirection to change the contents of the address. The value of the argument is stored temporarily in a local storage area for the called function.

## Examples

The following statement calls the function `startup`:

```
startup();
```

The following function call causes copies of `a` and `b` to be stored in a local area for `sum`. The function `sum` will then execute.

```
sum(a, b);
```

The following function call passes the value of 2 and the value of the expression `a + b` to `sum`:

```
sum(2, a + b);
```

The following statement calls the functions `printf` and `sum`. `sum` receives the values of `a` and `b`. `printf` receives a character string and the return value of the function `sum`:

```
printf("sum = %d\n", sum(a,b));
```

---

The following program passes the value of count to the function increment. increment increases the value of the parameter x by 1.

```
#include <stdio.h>
```

```
main()
{
    void increment();
    int count = 5;

    increment(count);
    printf("count = %d\n", count);
}
```

```
void increment(x)
int x;
{
    ++x;
    printf("x = %d\n", x);
}
```

The output illustrates that the value of count in main remains unchanged:

```
x = 6
count = 5
```

---

In the following program main passes the address of count to increment. The function increment was changed to handle the pointer. The parameter x is declared to be a pointer. The contents of what x points to is then incremented.

```
#include <stdio.h>
```

```
main()
{
    void increment();
    int count = 5;

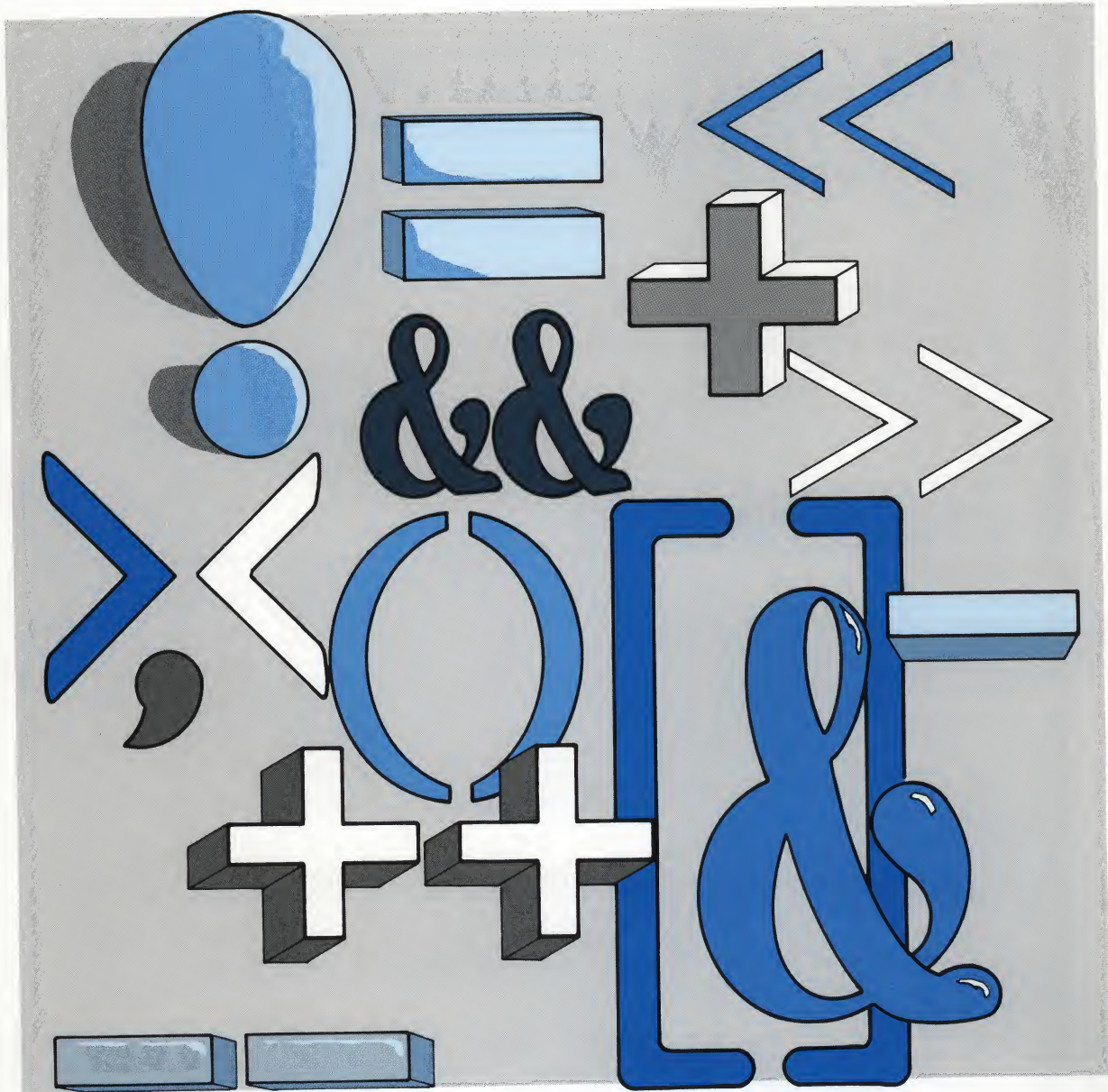
    increment(&count);
    printf("count = %d\n", count);
}
```

```
void increment(x)
int *x;
{
    ++*x;
    printf("*x = %d\n", *x);
}
```

The output shows that the variable count is incremented:

```
*x = 6
count = 6
```

---



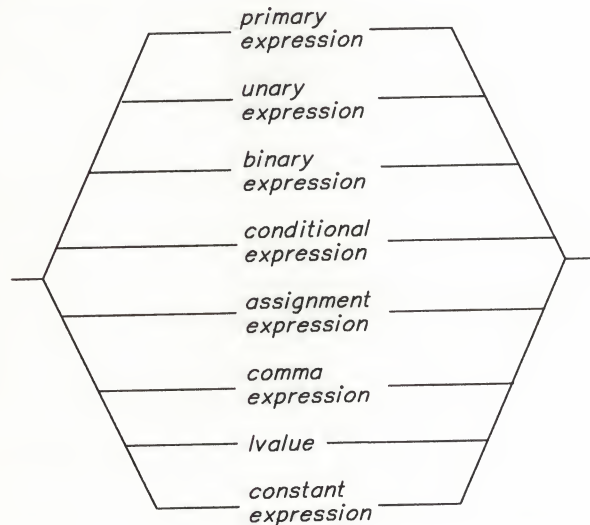
## CONTENTS

About This Chapter .....	13-3
Grouping and Evaluating Expressions .....	13-4
Lvalue .....	13-7
Constant Expression .....	13-9
Primary Expression .....	13-11
Parenthesized Expressions ( ) .....	13-12
Function Calls ( ) .....	13-12
Array Subscripts [ ] .....	13-14
Structure and Union Member Specifications . -> .....	13-15
Unary Expression .....	13-16
Increment ++ .....	13-17
Decrement -- .....	13-17
Arithmetic Negation - .....	13-18
Logical Negation ! .....	13-19
Bitwise Negation ~ .....	13-19
Address & .....	13-19
Indirection * .....	13-20
Cast (type name) .....	13-20
Size of an Object sizeof .....	13-21
Binary Expression .....	13-22
Multiplication * .....	13-23
Division / .....	13-23
Remainder % .....	13-23
Addition + .....	13-24
Subtraction - .....	13-24
Bitwise Left and Right Shift << >> .....	13-25
Relational < > <= >= .....	13-26
Equality == != .....	13-27
Bitwise AND & .....	13-28
Bitwise Exclusive OR ^ .....	13-28
Bitwise Inclusive OR   .....	13-29
Logical AND && .....	13-29
Logical OR    .....	13-30
Conditional Expression ? : .....	13-31
Assignment Expression .....	13-33
Simple Assignment = .....	13-33
Compound Assignment += -= *= /= %= <<= >>= &= ^=  = .....	13-34
Comma Expression , .....	13-35

## About This Chapter

This chapter describes C language expressions. Expressions are arranged in the following groups on the basis of the operators that the expressions contain and the contexts where the expressions can be used:

*expression*



Most expressions can contain several different, but related, types of operands. The following **type classes** describe related types of operands:

**Aggregate.** Arrays, structures, and unions.

**Scalar.** Arithmetic objects, and pointers to objects of any type.

**Arithmetic.** Integral objects and objects having the type **float** or **double**. The IBM RT PC C Language compiler also recognizes objects having the type **long double** as arithmetic objects.

**Integral.** Character objects, objects having an enumeration type, and objects having the type **short**, **int**, **long**, **unsigned short**, **unsigned int**, or **unsigned long**.

**Character.** Objects having the type **char** or **unsigned char**.

---

# Grouping and Evaluating Expressions

---

## Description

Two operator characteristics determine how operands group with operators: **precedence** and **associativity**. Precedence provides a priority system for grouping different types of operators with their operands. Associativity provides a left-to-right or right-to-left order for grouping operands to operators that have the same precedence. You can explicitly state the grouping of operands with operators by using parentheses.

The following table lists the C language operators in their order of precedence and shows the direction of associativity for each operator. The primary operators have the highest precedence. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

Precedence Level	Associativity	Operators
Primary	→	() [] . ->
Unary	←	++ -- - ! ~ & * ( <i>typename</i> ) sizeof
Multiplicative	→	* / %
Additive	→	+ -
Bitwise Shift	→	<< >>
Relational	→	< > <= >=
Equality	→	== !=
Bitwise Logical AND	→	&
Bitwise Exclusive OR	→	^
Bitwise Inclusive OR	→	
Logical AND	→	&&
Logical OR	→	

Figure 13-1 (Part 1 of 2). Operator Precedence and Associativity

Precedence Level	Associativity	Operators
Conditional	←	? :
Assignment	←	= += -= *= /= <<= >>= %= &= ^=  =
Comma	→	,

**Figure 13-1 (Part 2 of 2). Operator Precedence and Associativity**

The C language does not specify the order of evaluation for function call arguments or for the operands of binary operators (see “Exceptions”). Thus, avoid writing such ambiguous expressions as:

```
z = (x * ++y) / f(y)
f(++i, x[i])
```

## Exceptions

The C language does not specify the order of grouping operands with operators in an expression that contains more than one instance of an operator that has both associative and commutative properties. The compiler can rearrange operands in such an expression, and not even parentheses can guarantee an order of grouping within the expression. The operators that have both associative and commutative properties are: \*, +, &, !, and ^.

The order of evaluation for the operands of the logical AND and the logical OR operators is always left-to-right. If the operand on the left side of a && operator evaluates to 0 (zero), the operator on the right side is not evaluated. If the operand on the left side of a || operator evaluates to nonzero, the operator on the right side is not evaluated.

## Examples

The parentheses in the following expressions explicitly show how the C language groups operands and operators. If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses.

```
(total = (4 + (5 * 3) ) )
(total = ( ( (8 * 5) / 10) / 3) )
(total = (10 + (5/3) ) )
```

---

The following expression contains operators that are both associative and commutative:

```
total = price + state_tax + city_tax;
```

Because the C language does not specify the order of grouping operands with operators that are both associative and commutative, a compiler could group the operands and operators in the following ways (as indicated by parentheses):

```
(total = (price + (state_tax + city_tax) ) )  
(total = ( (price + state_tax) + city_tax) )  
(total = ( (price + city_tax) + state_tax) )
```

In the previous example the grouping of operands and operators does not affect the result. However, the grouping of operands and operators could affect the result in the following expression:

```
a = b() + c() + d()
```

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression:

```
x = b() + c()  
a = x + d()
```

## Related Information

“Parenthesized functions” on page 13-12.

---

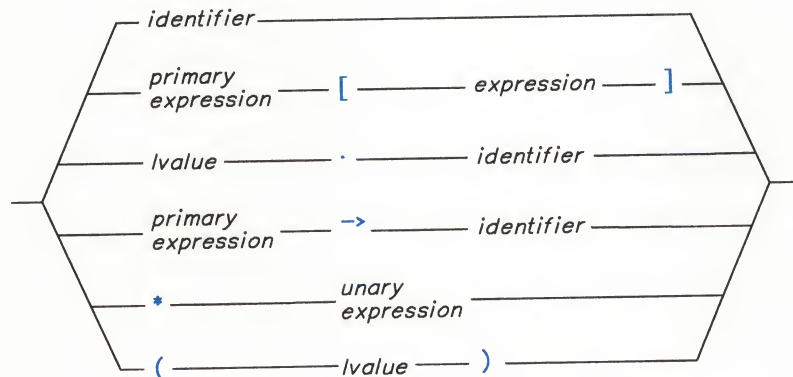
# Lvalue

---

## Description

An *lvalue* is an expression representing a data object that can be both examined and altered. An lvalue has the form:

*lvalue*



## Usage

The term *lvalue* comes from the lvalue's usage as the *left* operand in an assignment expression. All assignment operators evaluate their right operand (`vrbl_x = 825`) and assign that value to their left operand (`vrbl_x = 825`). The left operand must evaluate to a reference to an object.

The assignment operators are not the only operators that require an operand be an lvalue. The address operator, the increment operator, and the decrement operators also require an lvalue as an operand.

---

## Related Information

“Assignment Expression” on page 13-33.

“Address” on page 13-19.

“Structure and Union Member Specifications” on page 13-15.

---

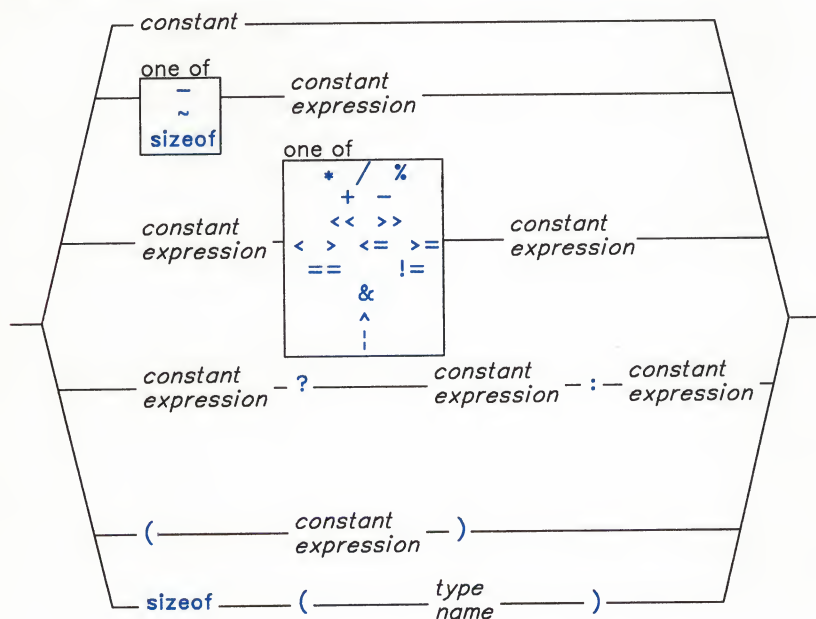
# Constant Expression

---

## Description

A **constant expression** is an expression with a value that is determined during compilation and cannot be changed during execution. A constant expression has the form:

*constant  
expression*



You can use parentheses to group expressions, but not to call functions.

---

## Usage

The C language requires constant expressions in the following places:

- In the subscript declarator, as the description of an array bound
- After the keyword **case** in a **switch** statement
- In the initializer of an external data definition
- In an enumerator, as the numeric value of an enum constant
- In a bit field-width specifier
- In the preprocessor **if** statement.

In a subscript declarator and in a **case** section of a **switch** statement, the constant expression can contain integer, character, and enumeration constants, casts to integral types, and **sizeof** expressions.

In an external data definition, the initializer must evaluate to a constant or to the address of an **extern** or **static** object (plus or minus an integer constant) that is defined or declared earlier in the file. Thus, the constant expression in the initializer can contain integer, character, enumeration, and float constants, casts to any type, **sizeof** expressions, and unary address expressions.

In a bit field-width specifier, the constant expression must evaluate to a non-negative value.

In a preprocessor **if** statement, the expression following **#if** can contain an integer constant, a character constant, or a preprocessor **defined** expression.

## Related Information

“Arrays” on page 11-8.

“External Data Definitions” on page 8-6.

“**switch**” on page 15-28.

“Enumerations” on page 11-17.

“Structures” on page 11-30.

“Conditional Compilation” on page 16-13.

---

# Primary Expression

---

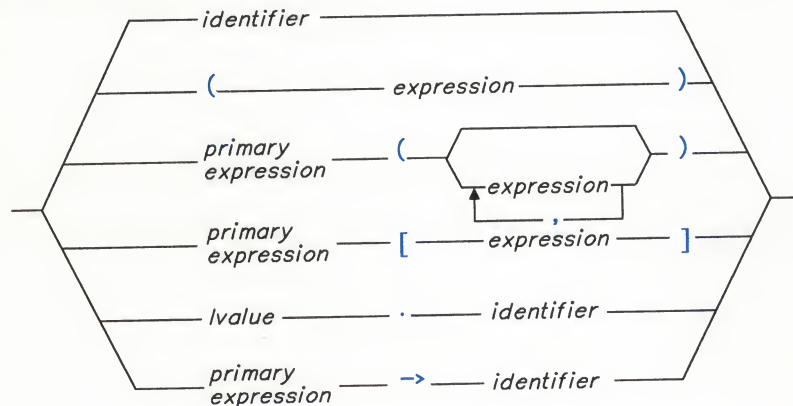
## Description

A **primary expression** can be:

- An identifier
- A parenthesized expression
- A function call
- An array element specification
- A structure or union member specification.

A primary expression has the form:

*primary  
expression*



All primary operators have the same precedence and have left-to-right associativity.

## Parenthesized Expressions ( )

You can use parentheses to explicitly state, and possibly change, how operands group with operators. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding `weight`, `zipcode` are used to form a function call. Notice how the compiler groups the operands and operators in this expression:

$$-\text{discount} * \text{item} + \text{handling}(\text{weight}, \text{zipcode}) < .10 * \text{item}$$

The following expression is similar to the previous expression. This expression, however, contains parentheses that change how the operands and operators group:

$(-\text{discount} * (\text{item} + \text{handling}(\text{weight}, \text{zipcode}))) < (.10 * \text{item})$

In an expression that contains operators that are both associative and commutative, the compiler can ignore parentheses that are intended to specify the grouping of operands with operators. The parentheses in the following expression do not guarantee the order of grouping operands with the operators:

$$x = f() + (g() + h())$$

## Function Calls ( )

A **function call** is a primary expression followed by a parenthesized argument list. The argument list can contain any number of expressions separated by commas. For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

The actual parameters are copied for transmission to the called function from an lvalue; all parameters are passed by value. Assigning to a parameter changes the value within the function, but has no effect on the lvalue.

In the following example, `main` passes `func` two values: 5 and 7. `func` receives these values and gives them the identifiers: `a` and `b`. `func` changes the value of `a`. When control passes back to `main`, the system frees the storage for `a` and `b`. The actual values of `x` and `y` never change.

```
main()
{
    int x = 5, y = 7, z;

    func(x, y);
    printf("In main, x = %d    y = %d\n", x, y);
}

func(a,b)
int a, b;
{
    a += b;
    printf("In func, a = %d    b = %d\n", a, b);
}
```

The preceding program produces the following output:

```
In func, a = 12    b = 7
In main, x = 5     y = 7
```

If you define a primary expression as *function returning type `int`*, the result of the function call has type `int`. If you define a primary expression as a function returning another type, the result of the function call has that other type. Function parameters cannot be an array or a function or have type `void`. Functions can return a value of any type except an array or a function.

A function can change the value of the object a pointer references. If you want a function to change a parameter, pass a pointer to the variable you want changed. When a pointer is passed as a parameter, the pointer itself is copied: the object pointed to is not copied. (See "Pointers" on page 11-22.)

The usual arithmetic conversions are carried out before a call (see "Usual Arithmetic Conversions" on page 14-5). Other conversions are not performed. Use a cast expression for other conversions (see "Cast" on page 13-20).

If the compiler cannot find a function definition that matches the primary expression, the compiler assumes the function to be of the external storage class, returning type `int`.

---

The compiler does not compare the data types that a function sends with the data types that the called function receives.

The C language does not define the order that the compiler evaluates parameters. Avoid such calls as:

```
method(process, batch.process--, batch.process)
```

In the preceding example, `batch.process--` may be processed last, causing two arguments to be passed with the same value.

A function can call itself.

The compiler may not issue a warning if the number of actual parameters does not match the number of formal parameters of the function being called; information about the formal parameters is not always available to the compiler.

See Chapter 12, “Functions” for detailed characteristics of functions.

## Array Subscripts [ ]

A primary expression followed by an expression in [ ] (square brackets) refers to an element of an array.

The primary expression must point to the first member of the array. The expression in brackets must be a **subscript** to the primary expression. The subscript must have an integral type. The type of the array determines the type of the result. For example, if `prime` has type array of `int`, then `prime[2][3]` produces a value of type `int`.

The first subscript of each dimension is 0 (zero). Thus, the expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of their increasing storage locations) by incrementing the right-most subscript most frequently. For example, the following statement gives the value 0 (zero) to each element in the array `code[4][3][6]`:

```
for (first = 0; first <= 3; ++first)
    for (second = 0; second <= 2; ++second)
        for (third = 0; third <= 5; ++third)
            code[first][second][third] = 0;
```

“Arrays” on page 11-8 explains how to define and use an array.

---

## Structure and Union Member Specifications . ->

Two primary operators enable you to specify structure and union members: . (dot) and -> (arrow).

The dot (a period) and arrow (formed by a minus and a greater than symbol) operators are always preceded by a primary expression and followed by an identifier.

When you use the dot operator, the primary expression must represent a structure or union and the identifier must name a member of that structure or union. The result is the value of the named structure or union member. The result is an lvalue if the first expression is an lvalue.

Some sample dot expressions follow:

```
roster[num].name  
roster[num].name[1]
```

When you use the arrow operator, the primary expression must be a pointer to a structure or a union and the identifier must name a member of the structure or union. The result is the value of the named structure or union member to which the pointer expression referred. The result is an lvalue if the member has an allowable lvalue type. For example:

```
roster -> name
```

See also “Defining New Data Types and New Names for Existing Data Types” on page 11-4 and “Structures” on page 11-30.

---

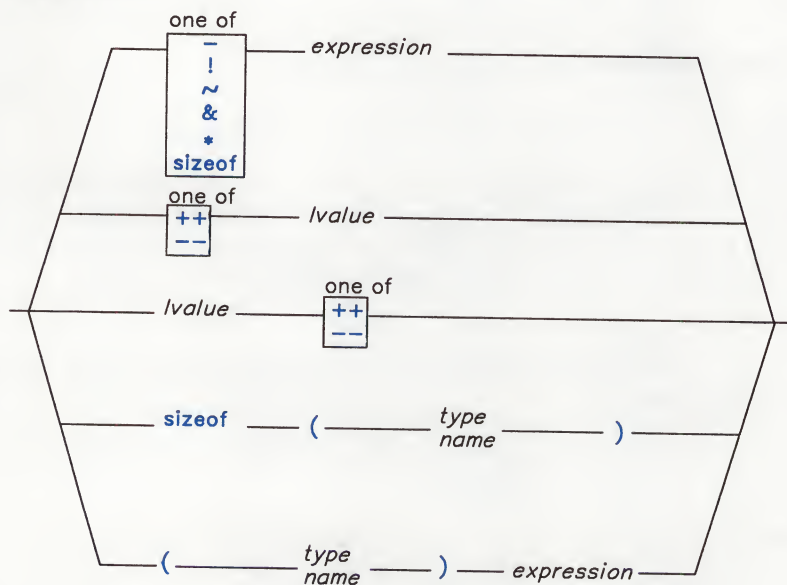
# Unary Expression

---

## Description

A **unary expression** contains one operand. A unary expression has the form:

*unary  
expression*



All unary operators have the same precedence and have right-to-left associativity. The following sections describe the unary operators and the operations they perform.

---

## Increment ++

The ++ (increment) operator adds 1 (one) to the value of the operand, or if the operand is a pointer, changes the value of the operand so that the operand points to the next address. The operand receives the result of the increment operation. Thus, the operand must be an lvalue. If the operand is a pointer, the operand must point to an object in an array.

You can place the ++ before or after the operand. If the ++ appears before the operand, the system increments the operand, then uses the incremented value in the expression. If you place the ++ after the operand, the system uses the current value of the operand in the expression, then increments the operand. For example:

```
play = ++play1 + play2++
```

is equivalent to the following three expressions:

```
play1 = play1 + 1  
play = play1 + play2  
play2 = play2 + 1
```

Because the C language does not specify the order of evaluation for many expressions, avoid using a variable more than once in an expression where the variable is incremented. For example, the following expression may cause *i* to be incremented before or after the function call to *x* is executed:

```
y = x(i) + i++
```

The compiler performs the usual arithmetic conversions on the operand and the assignment conversion on the result of the operation. See “Usual Arithmetic Conversions” on page 14-5 and “Assignment Conversion” on page 14-7.

## Decrement --

The -- (decrement) operator subtracts 1 (one) from the value of the operand, or if the operand is a pointer, changes the value of the operand so that the operand points to the preceding address. The operand receives the result of the decrement operation. Thus, the operand must be an lvalue. If the operand is a pointer, the operand must point to an object in an array.

You can place the -- before or after the operand. If the -- appears before the operand, the system decrements the operand, then uses the decremented value in the expression. If the -- appears after the operand, the system uses the current value of the operand in the expression, then decrements the operand.

---

For example:

```
play = --play1 + play2--
```

is equivalent to the following three expressions:

```
play1 = play1 - 1  
play = play1 + play2  
play2 = play2 - 1
```

Because the C language does not specify the order of evaluation for many expressions, avoid using a variable more than once in an expression where the variable is decremented. For example, the following expression may cause *i* to be decremented before or after the function call to *x* is executed:

```
y = x(i) + i--
```

The compiler performs the usual arithmetic conversions on the operand and the assignment conversion on the result of the operation. See “Usual Arithmetic Conversions” on page 14-5 and “Assignment Conversion” on page 14-7.

## Arithmetic Negation -

The **-** (arithmetic negation) operator negates the value of the operand. The value of the operand can have any arithmetic type (**char**, **int**, **float**, **double**, or **enum**).

The negation of a signed operand is:

*0 - operand*

For example, if *quality* has the value 100, then *-quality* has the value -100 or:

*0 - 100*

The negation of an unsigned operand is:

*(~operand) + 1*

The system performs the usual arithmetic conversions on the operand. See “Usual Arithmetic Conversions” on page 14-5.

---

## Logical Negation !

The ! (logical negation) operator determines whether the operand evaluates to 0 (zero). If so, the operation yields the value 1 (one). Otherwise, the operation yields the value 0 (zero). The operand must have a scalar data type, but the result of the operation is always type `int`.

The following two expressions are equivalent:

```
!right  
right == 0
```

## Bitwise Negation ~

The ~ (bitwise negation) operator yields the ones complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type.

The language does not specify the result of the bitwise negation operation when the operand has a signed type.

Suppose `x` represents the decimal value 5. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `~x` yields the following result (represented here as a 16-bit binary number):

```
111111111111010
```

The 16-bit binary representation of `~0` is:

```
111111111111111
```

## Address &

The & (address) operator yields a pointer to the location of its operand. The operand must be an lvalue and cannot have the storage class **register**. The type of the operand determines the type of the result. Thus, if the operand has type `int`, the result is a pointer to an object having type `int`.

If `p-to-y` is defined as a pointer to an `int` and `y` as an `int`, the expression:

```
p-to-y = &y
```

assigns the address of the variable `y` to the pointer `p-to-y`.

See also "Pointers" on page 11-22.

---

## Indirection \*

The \* (indirection) operator determines the value referred to by the pointer-type operand. The operation yields an lvalue. The system performs the usual unary conversions on the operand, with arrays and functions being converted to pointers. The type of the operand determines the type of the result. Thus, if the operand is a pointer to an **int**, the result has type **int**.

The language does not specify the results of applying the indirection operator to a null pointer.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the expressions:

```
p_to_y = &y  
*p_to_y = 3
```

cause the variable `y` to receive the value 3.

See also "Pointers" on page 11-22.

## Cast (*type name*)

A **cast** operator converts the value of the operand to a specified data type. The cast operator is a parenthesized name of a data type. This data type, to which the operand is converted, must be scalar. Chapter 14, "Conversions" describes how the C language performs conversions.

The following expression contains the `(double)x` cast expression:

```
printf("x=%f\n", (double)x)
```

The function `printf` receives the value of `x` as a **double**. The variable `x` remains unchanged by the cast.

---

## Size of an Object    `sizeof`

The **sizeof** operator yields the size in *bytes* of the operand. The system must be able to evaluate the size at compile time. The language does not specify the size of a byte. However, most C language compilers (including the IBM RT PC C Language compiler) consider the size of a **char** object as the size of a byte. Given that the variable *x* has type **short** and that the size of a **short** is 2 bytes, the following expression evaluates to 2:

```
sizeof (x)
```

The type of the result is usually an **unsigned int** or an **unsigned long**.

The compiler determines the size of an object on the basis of the object definition. The **sizeof** operator does not perform any conversions. However, if the operand contains operators that perform conversions, the compiler takes these conversions into consideration. The following expression causes the compiler to perform the usual arithmetic conversions. The result of this expression has type **int** (if *x* has type **char**, **short**, or **int** or any enumeration type):

```
sizeof (x + 1)
```

When you perform the **sizeof** operation on an array, the result is the total number of bytes in the array. The compiler does not convert the array to a pointer before evaluating the expression.

All **sizeof** operations are performed at compile time. Therefore, the following expression will not increment *x* by 5:

```
sizeof (x += 5)
```

You can use a **sizeof** expression anywhere a constant or unsigned constant is required. One of the most common usages for the **sizeof** operator is to determine the size of objects that are being communicated to or from storage allocation, input, and output functions.

You can use the **sizeof** operator to determine the size that a data type represents. In this case, the name of the data type must be placed in parentheses after the **sizeof** operator. For example:

```
sizeof(int)
```

The compiler reads the expression `sizeof(total)` as a unit. Thus, the following two expressions yield the same value:

```
sizeof(total) - 2  
(sizeof(total)) - 2
```

---

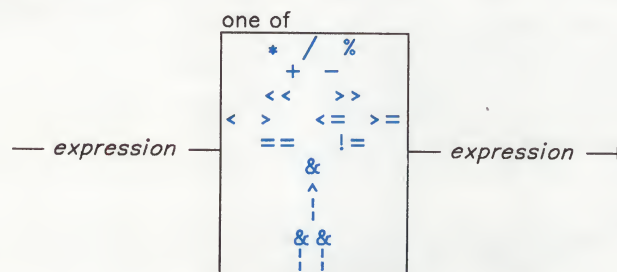
# Binary Expression

---

## Description

A **binary expression** contains two operands separated by one operator. A binary expression has the form:

*binary  
expression*



Not all binary operators have the same precedence. The preceding binary expression diagram shows the order of precedences among binary operators. In the box that lists the operators, each row represents a different level of precedence. The top row has the highest precedence, and the bottom row has the lowest precedences (among binary operators). All binary operators have left-to-right associativity.

The language does not specify the order in which the compiler evaluates the operands of most binary operators. Therefore, to ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

The following sections describe the binary operators and the operations they perform.

---

## Multiplication \*

The \* (multiplication) operator yields the product of its operands. The operands must have an arithmetic type. The compiler performs the usual arithmetic conversions on the operands. See "Usual Arithmetic Conversions" on page 14-5.

Because the multiplication operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one multiplication operator, even when the sub-expressions are explicitly grouped with parentheses. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
```

```
sites * (number * cost)
```

```
(cost * sites) * number
```

## Division /

The / (division) operator yields the quotient of its operands. The operands must have an arithmetic type. If both operands are positive and the operation produces a remainder, the compiler ignores the remainder. Thus, the expression  $7 / 4$  yields the value 1 (rather than 1.75 or 2).

The language does not define how the compiler treats the remainder when either of the operands has a negative value. Thus,  $-7 / 4$  can yield either -1 or 2. The IBM RT PC C Language evaluates  $-7 / 4$  to -1.

The result is undefined by the language if the second operand evaluates to 0 (zero).

The compiler performs the usual arithmetic conversions on the operands. See "Usual Arithmetic Conversions" on page 14-5.

## Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression  $5 \% 3$  yields 2.

Both operands must have an arithmetic type. If the right operand evaluates to 0 (zero) or if either operand has a negative value, the result is undefined. The result has the same sign as the left operand.

---

The following expression always yields the value of *a* if *b* is not 0 (zero):

`( a / b ) * b + a % b`

The compiler performs the usual arithmetic conversions on the operands. See “Usual Arithmetic Conversions” on page 14-5.

## Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must have an integral type and the other must have a pointer type. (A pointer must point to an element in an array.)

When both operands have an arithmetic type, the compiler performs the usual arithmetic conversions on the operands. The result has the type produced by the conversions on the operands.

When one of the operands is a pointer, the compiler converts the other operand to an address offset. (See “Integral to Pointer” on page 14-9.) The result is a pointer of the same type as the pointer operand.

## Subtraction -

The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral type. (A pointer must point to an element in an array.)

When both operands have an arithmetic type, the compiler performs the usual arithmetic conversions on the operands. The result has the type produced by the conversions on the operands.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right operand to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to the same type, the compiler converts the result to an `int` that represents the number of objects separating the two addresses. The language does not define the behavior of the compiler if the pointers do not refer to objects in the same array.

---

## Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit positions of a binary value. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted.

The << (bitwise left shift) operator indicates the bits are to be shifted to the left. The >> (bitwise right shift) operator indicates the bits are to be shifted to the right.

Each operand must have an integral type. The compiler performs the usual arithmetic conversions on the operands. Then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions occur).

If the right operand has a negative value or a value that is greater than or equal to the width in bits of the expression being shifted, the result is undefined.

If the right operand has the value 0 (zero), the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `l_op` has the value 25, the bit pattern (in 16-bit format) of `l_op` is:

```
0000000000011001
```

If `r_op` has the value 3, the expression `l_op << r_op` yields:

```
0000000011001000
```

If the left operand has an **unsigned** type, the >> operator fills vacated bits with zeros. Otherwise, the language does not specify how the vacated bits produced by the >> operator are filled. The compiler can fill the vacated bits of a signed value with a copy of the value's sign bit (as is done by the IBM RT PC C Language compiler) or with zeros. For example, if `l_op` has the value -25, the bit pattern (in 16-bit format) of `l_op` is:

```
111111111100111
```

If `r_op` has the value 3 and the system fills vacated bits with a copy of the sign bit, the expression `l_op >> r_op` yields:

```
111111111111100
```

If the system fills vacated bits with zeroes, the expression `l_op >> r_op` yields:

```
000111111111100
```

---

## Relational   < > <= >=

The relational operators compare two operands for the validity of a relationship. If the relationship stated by the operator is true, the value of the result is 1 (one). Otherwise, the value of the result is 0 (zero).

The following table describes the relational operators:

Operator	Usage
<	Indicates whether the value of the left operand is less than the value of the right operand.
>	Indicates whether the value of the left operand is greater than the value of the right operand.
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type. The result has type **int**.

If the operands have arithmetic types, the compiler performs the usual arithmetic conversions on the operands.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

Relational operators have left-to-right associativity. Therefore, the expression:

`a < b <= c`

is interpreted as:

`(a < b) <= c`

If the value of `a` is less than the value of `b`, the first relationship is true and yields the value 1 (one). The compiler then compares the value 1 (one) with the value of `c`.

---

## Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship stated by an equality operator is true, the value of the result is 1 (one). Otherwise, the value of the result is 0 (zero).

The following table describes the equality operators:

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type, or one operand must have a pointer type and the other operand must have an integral type and evaluate to 0 (zero). The result has type `int`.

If the operands have arithmetic types, the compiler performs the usual arithmetic conversions on the operands.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

If one operand is a pointer and the other operand is an integer having the value 0 (zero), the expression is true only if the pointer operand evaluates to a null pointer.

The following expressions contain equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

---

## Bitwise AND &

The & (bitwise AND) operator compares the values (in binary format) of each operand and yields a value with a bit pattern that shows which bits in each operand have the value 1 (one). Each bit that evaluates to 1 (one) in both operands receives the value 1 (one) in the result. All other bits receive the value 0 (zero).

Both operands must have an integral type. The compiler performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands.

Because the bitwise AND operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise AND operator, even when the sub-expressions are explicitly grouped with parentheses.

The following example shows the values of a, b, and the result of a & b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a & b	0000000000001100

## Bitwise Exclusive OR ^

The ^ (bitwise exclusive OR) operator compares the values (in binary format) of each operand and yields a value with a bit pattern that shows which bits in one operand (not both operands) has the value 1 (one). Each bit that evaluates to 1 (one) in only one operand receives the value 1 (one) in the result. All other bits receive the value 0 (zero).

Both operands must have an integral type. The compiler performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands.

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise exclusive OR operator, even when the sub-expressions are explicitly grouped with parentheses.

The following example shows the values of a, b, and the result of a ^ b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a ^ b	0000000001110010

## Bitwise Inclusive OR |

The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1 (one). Each bit that evaluates to 1 (one) in either operand receives the value 1 (one) in the result. All other bits receive the value 0 (zero).

Both operands must have an integral type. The compiler performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise inclusive OR operator, even when the sub-expressions are explicitly grouped with parentheses.

The following example shows the values of a, b, and the result of a | b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a   b	0000000001111110

## Logical AND &&

The && (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value 1 (one). Otherwise, the result has the value 0 (zero).

Both operands must have a scalar type. The compiler performs the usual arithmetic conversions on each operand. The result has type **int**.

The logical AND operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (zero), the right operand is not evaluated.

The following examples show how the language evaluates expressions that contain the logical AND operator:

Expression	Result
1 && 0	0
1 && 4	1
0 && 0	0

---

The following example uses the logical AND operator to avoid a divide-by-zero situation:

```
y != 0 && x / y
```

The system does not evaluate the expression `x / y` when the expression `y != 0` evaluates to 0 (zero).

## Logical OR `||`

The `||` (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value 1 (one). Otherwise, the result has the value 0 (zero).

Both operands must have a scalar type. The compiler performs the usual arithmetic conversions on each operand. The result has type `int`.

The logical OR operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, the system does not evaluate the right operand.

The following examples show how the language evaluates expressions that contain the logical OR operator:

Expression	Result
<code>1    0</code>	1
<code>1    4</code>	1
<code>0    0</code>	0

The following example uses the logical OR operator to conditionally increment `y`:

```
++x || ++y
```

The system does not evaluate the expression `++y` when the expression `++x` evaluates to a nonzero quantity.

---

## Conditional Expression ? :

---

### Description

A **conditional expression** is a compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value 0 (zero). A conditional expression has the form:

*conditional  
expression*

*— expression — ? — expression — : — expression —*

The conditional expression contains one two-part operator. The ? symbol follows the condition, and the : symbol appears between the two action expressions. The compiler treats all expressions that occur between the operators ? and : as one expression.

The left operand can have any type. The second and third operands must have arithmetic types, the same structure type, or the same union type. Otherwise, the second and third operands must be pointers to the same type, or one operand must be a pointer and the other operand must be the constant 0 (zero).

The system performs the usual arithmetic conversions on the second and third expressions, if these expressions have arithmetic types. Then the compiler evaluates the first expression. If the first expression has a nonzero value, the system evaluates the second expression. Otherwise, the system evaluates the third expression.

The types of the second and third operands determine the type of the result. If these operands have arithmetic types, the result has the type produced by the arithmetic conversions. If these operands are structures or unions of *type* or pointers to *type*, the result has *type*. If the second operand is a pointer and the third operand is the constant 0 (zero), the result is a null pointer.

Conditional expressions have right-to-left associativity. The system evaluates the left operand first, then only one of the remaining two operands.

---

## Examples

The following expression determines which variable has the greater value, `y` or `z`, and assigns the greater value to the variable `x`:

```
x = (y > z) ? y : z
```

The preceding expression is equivalent to the following statement:

```
if (y > z)
    x = y;
else
    x = z;
```

The following expression calls the function **printf**. **printf** receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, **printf** receives the character constant `'x'`.

```
printf(" c = %c\n", (c >= '0' && c <= '9') ? c : 'x')
```

---

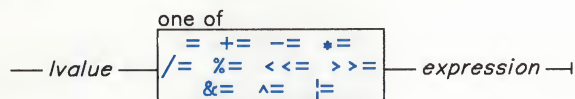
# Assignment Expression

---

## Description

An **assignment expression** gives a value to the left operand. An assignment expression has the form:

*assignment  
expression*



The left operand in all assignment expressions must be an lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

All assignment operators have the same precedence and have right-to-left associativity.

The language contains two types of assignment operators, simple assignment and compound assignment operators. The following sections describe these operators.

## Simple Assignment =

The simple assignment operator gives the value of the right operand to the left operand.

Both operands must have arithmetic types, the same structure type, or the same union type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant 0 (zero).

If both operands have arithmetic types, the system converts the value of the right operand to the type of the left operand before the assignment.

If the left operand is a pointer and the right operand is the constant 0 (zero), the result is a null pointer.

The following example assigns the value of number to the member employee of the structure payroll:

```
payroll.employee = number
```

---

The following example assigns the value 0 (zero) to d, the value of d to c, the value of c to b, and the value of b to a:

```
a = b = c = d = 0
```

## Compound Assignment   +=   -=   \*=   /=   %=   <<=   >>=   &=   ^=   !=

The compound assignment operators perform an operation on both operands and give the result of that operation to the left operand.

The left operand of the += and -= operators must be a pointer and the right operand must have an integral type, or both operands must have an arithmetic type.

Both operands of the \*=, /=, and %= must have an arithmetic type.

Both operands of the <<=, >>=, &=, ^=, and != operators must have an integral type.

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
+=	index += 2	index = index + 2
-=	*(pointer++) -= 1	*pointer = *(pointer++) - 1
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2
^=	test ^= pre_test	test = test ^ pre_test
!=	flag != ON	flag = flag != ON

Although the equivalent expression column shows the left operands (from the example column) evaluated twice, the compiler may evaluate the left operand once or several times.

---

# Comma Expression ,

---

## Description

A **comma expression** contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. If the left operand produces a value, the compiler discards this value. A comma expression has the form:

*comma  
expression*

*— expression —, — expression —*

Both operands of a comma expression can have any type. All comma expressions have right-to-left associativity. The compiler evaluates the left operand before the right operand.

If omega had the value 11, the following example would increment y and assign the value 3 to alpha:

```
alpha = (y++, omega % 4)
```

Any number of expressions separated by commas can form a single expression. The compiler evaluates the leftmost expression first. The value of the rightmost expression becomes the value of the entire expression. For example, the value of the following expression is the value of the expression rotate(direction):

```
intensity++, shade * increment, rotate(direction)
```

## Restrictions

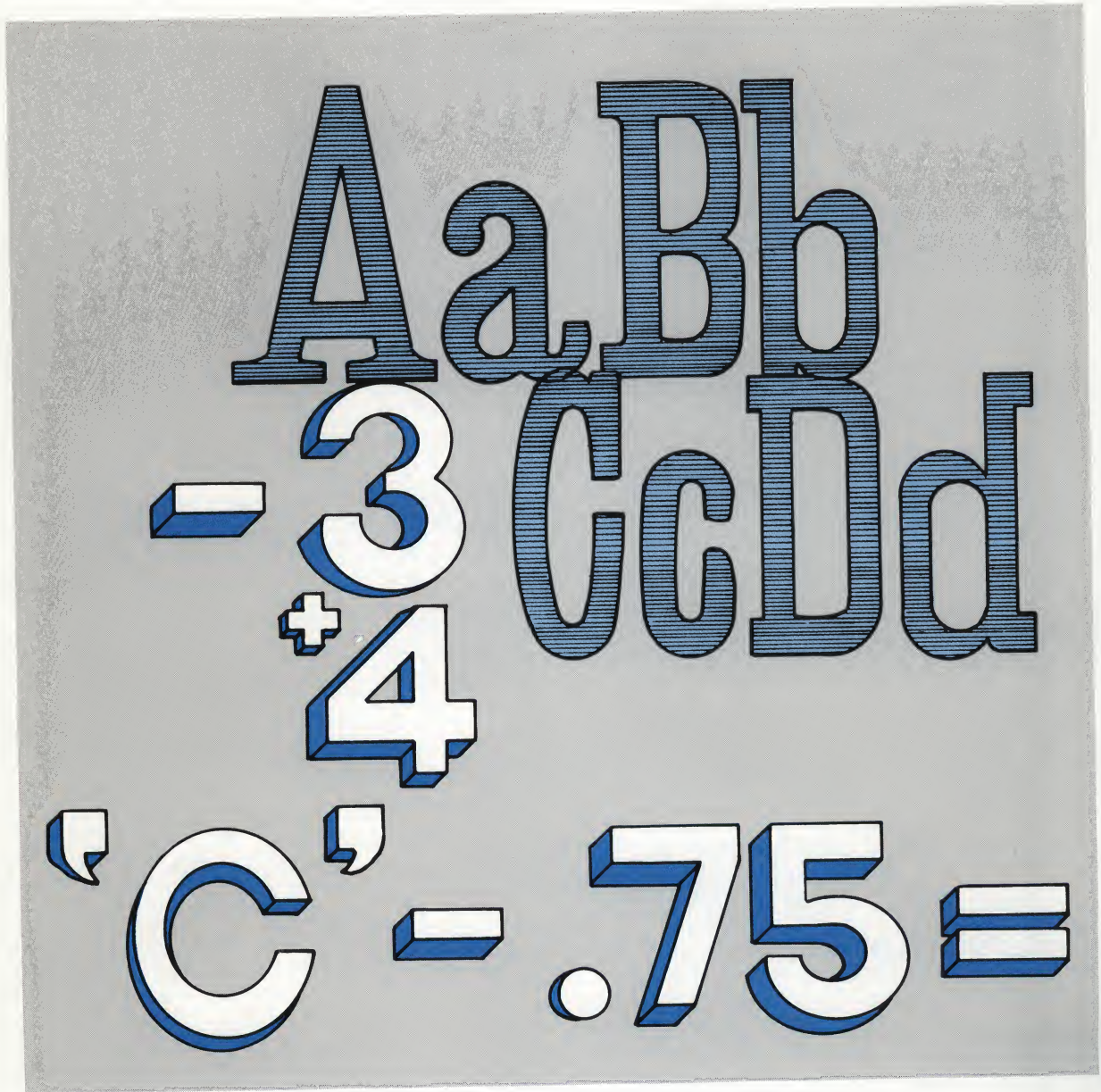
You can place comma expressions within lists that contain commas (for example, argument lists and initializer lists). However, because the comma has a special meaning, you must place parentheses around comma expressions in these lists. The following function call contains the comma expression `t = 3, t + 2`:

```
f(a, (t = 3, t + 2), c)
```

The arguments to the function f are: the value of a, the value 5, and the value of c.



## Chapter 14. Conversions



---

## CONTENTS

About This Chapter .....	14-3
Usual Unary Conversions .....	14-4
Usual Arithmetic Conversions .....	14-5
Widening .....	14-5
Type Balancing .....	14-6
Sign Balancing .....	14-6
Assignment Conversion .....	14-7
Explicit Conversions .....	14-8
Reduction Conversions .....	14-8
Expansion Conversions .....	14-9
Pointer Conversions .....	14-9
<b>void</b> Conversions .....	14-10
<b>volatile</b> Conversions .....	14-10

---

## About This Chapter

Many C language operators cause **conversions**. A conversion changes the form of a value. For example, when you add values having different data types, the compiler converts both values to the same form before adding the values. Addition of a **short int** value and an **int** value causes the compiler to convert the **short int** value to the **int** form.

Conversions may occur when:

- A cast operation is performed.
- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A function is provided a value that has a different type than the parameter.
- A function returns a value that has a different type than the defined return type for the function.

Although the language contains some guidelines for conversions, many conversions contain implementation specific aspects. These implementation specific aspects occur because:

- The sizes of the data types vary.
- The manner of handling signed data varies.
- The data formats vary.

---

## Usual Unary Conversions

---

The *usual unary conversions* reduce the types of values that the compiler must handle. The compiler uses the usual unary conversions on:

- The operands of the unary operators `!`, `-`, `~`, and `*`
- The operands of the binary operators `<<` and `>>`
- The arguments in a function call (before the function is called).

The following table lists the types of values that the usual unary conversions affect:

Type of Value Before Conversion	Type of Value After Conversion
<code>char</code>	<code>int</code>
<code>unsigned char</code>	<code>unsigned int</code>
<code>short</code>	<code>int</code>
<code>unsigned short</code>	<code>unsigned int</code>
<code>float</code>	<code>double</code> <sup>1</sup>
<code>array of <i>type</i></code>	<code>pointer to <i>type</i></code>

Figure 14-1. Usual Unary Conversions

“Explicit Conversions” on page 14-8 describes how the compiler performs conversions.

---

<sup>1</sup> The IBM RT PC C Language compiler performs the usual unary conversion of `float` to `double` on arguments in function calls only. When a `float` object appears as an operand of `!`, `-`, `~`, `*`, `<<`, or `>>`; the IBM RT PC C Language compiler does not perform a usual unary conversion.

---

## Usual Arithmetic Conversions

---

The **usual arithmetic conversions** reduce the types of objects that the compiler handles when performing arithmetic operations. Many compilers perform arithmetic operations only on objects having one of several data types. These types on the IBM RT PC C Language compiler are: **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double**, and **long double**. If all operands do not have one of these types, the system converts the values of the operands according to the following procedures:

1. **Widening** values that do not have data types appropriate for arithmetic operations.
2. **Type balancing** values in operations that have more than one operand.
3. **Sign balancing** values in operations that have more than one operand.

The following sections describe the usual arithmetic conversion procedures.

### Widening

**Widening** expands the size of a value (for example, **short** to **int** by padding bits located to the left of the value with a copy of the sign bit). Widening does not affect the sign of the value.

The following table shows the types of values that the compiler widens:

Type of Value Before Widening	Type of Value After Widening
<b>char</b>	<b>int</b> or <b>unsigned int</b>
<b>unsigned char</b>	<b>unsigned int</b>
<b>short</b>	<b>int</b>
<b>unsigned short</b>	<b>unsigned int</b>
<b>float</b>	<b>double</b>

Figure 14-2. Arithmetic Widening Conversions

The IBM RT PC C Language compiler treats **char** objects as **unsigned** values. Thus, widening of a **char** yields an **int** that has a positive value.

Many compilers widen **float** values to **double** values before performing arithmetic operations. Where possible, the IBM RT PC C Language compiler performs single precision arithmetic on **float** values.

---

## Type Balancing

*Type Balancing* makes all operands have the same size data type. If both of the operands do not have the same size data type, the compiler converts the value of the operand having the smaller type to a value having the larger type.

For example, if the operand `count` had type `int` and the operand `maximum` had type `long`, the compiler would convert the value of `count` to type `long`.

Type balancing does not affect the sign of the value.

## Sign Balancing

*Sign balancing* makes both operands have the same data type (signed or **unsigned**). If one operand has an **unsigned** type, the compiler converts the other operand to that **unsigned** type. Otherwise, both operands remain signed.

---

## Assignment Conversion

---

An **assignment conversion** makes the value of the right operand have the same data type as the left operand. Only the following assignment type combinations are supported by the language:

Type of Left Operand	Type of Right Operand
Any arithmetic type	Any arithmetic type
Pointer to <i>type</i>	Pointer to <i>type</i> OR the constant 0 (zero)
Structure of <i>type</i>	Structure of <i>type</i>
Union of <i>type</i>	Union of <i>type</i>

**Figure 14-3. Assignment Conversions**

The constant 0 (zero) assigned to a pointer causes the compiler to convert the value 0 to a NULL pointer.

“Explicit Conversions” on page 14-8 describes how the compiler performs conversions from one arithmetic type to another arithmetic type.

---

## Explicit Conversions

---

When the compiler converts the value of one data type to the value of another data type, the compiler usually performs one of the following three types of conversions:

**Reduction conversions.** Change the data type of a value to a smaller size data type (for example, a value having type **double** to a value having type **float**).

**Expansion conversions.** Change the data type of a value to a larger size data type (for example, a value having type **float** to a value having type **double**).

**Pointer conversions.** Change the data type to which a pointer refers or change an integral type to a pointer.

**void conversions.** Discard the value of a function call.

**volatile conversions.** Give a non-volatile data object the **volatile** attribute.

The following sections describe these conversions.

### Reduction Conversions

#### Integral Reduction

When the compiler converts an integral value to a narrower type (for example, a **long** to a **short**), the compiler truncates the value by discarding the most significant bits.

#### double or long double to float

When the compiler converts a double precision floating-point value (**long double** or **double**) to a single precision floating-point value (**float**), the compiler rounds off the double precision value.

#### Floating-Point to Integral

The language does not define the method of converting floating-point values to integral values. The IBM RT PC C Language compiler drops the fraction part of the floating-point value.

#### Integral to Floating-Point

The C language does not prohibit integral sizes from having a higher precision than the floating-point sizes. If a higher precision integer is converted to a **float**, the resulting **float** may experience a loss of precision.

---

## Expansion Conversions

### Floating-Point Expansion

Although many compilers perform all floating-point arithmetic in double precision only, the IBM RT PC C Language compiler performs single precision arithmetic when all operands have type **float**. When one operand has type **float** and another operand has type **double** or **long double**, the IBM RT PC C Language compiler converts the **float** value to the equivalent **double** or **long double** value.

### Integral to Floating-Point

The compiler converts narrower integral values to equivalent floating-point values.

### Unsigned Arithmetic Expansion

The compiler converts narrower **unsigned** arithmetic values to wider **unsigned** arithmetic values by padding the values with zeros.

### Signed Arithmetic Expansion

The language does not define how narrower signed arithmetic values are converted to wider signed arithmetic values. When a narrower arithmetic value is converted to a wider signed arithmetic value, the compiler pads bits located to the left of the value with a copy of the sign bit.

## Pointer Conversions

### Pointer to Pointer

When two pointers to objects of the same type are added or subtracted, the compiler performs the indicated operation (addition or subtraction) on the values of the pointers and divides the result by the length of the objects to which the pointers refer. The result is an integer that indicates the distance between the specified objects in the array. For example, if *p* points to the second element in an array and *q* points to the fifth element in the same array, the expression *p - q* yields -3.

### Integral to Pointer

When an integral value is added to or subtracted from a pointer, the compiler multiplies the integral value by the length of the object to which the pointer refers to produce an address offset, which can be added to or subtracted from the pointer value. The result is a pointer (having the same type as the original pointer) that refers to an object assumed to be in the same array.

---

## void Conversions

A program cannot use or apply conversions to the (nonexistent) value of a **void** object. You can convert the result of a function call to type **void** by using the cast operator. Such a conversion discards the value of a function call used in an expression statement. For example, the following statement discards the result of the function call `add()`:

```
(void)add();
```

## volatile Conversions

### volatile to Non-volatile

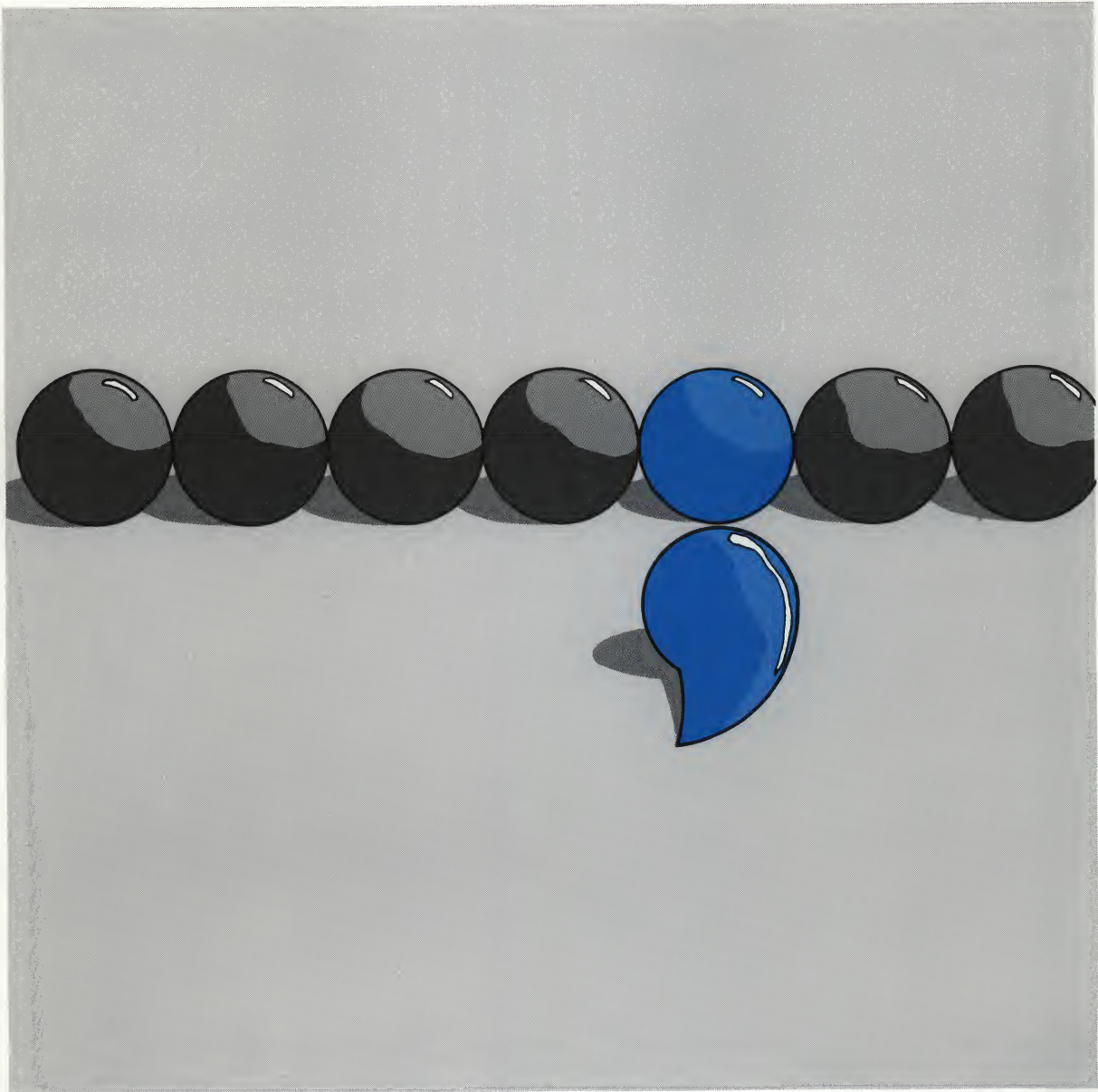
Through an explicit cast, you can assign the address of a **volatile** data object to a pointer that is defined as pointing to a non-**volatile** data object. If the **volatile** object is referenced through such a pointer, the result is undefined.

### Non-volatile to volatile

You can assign the address of a non-**volatile** data object to a pointer that is defined as pointing to a volatile data object. If the non-**volatile** object is referenced through such a pointer, the compiler treats the non-**volatile** object as a **volatile** object.

---

## Chapter 15. C Language Statements



---

## CONTENTS

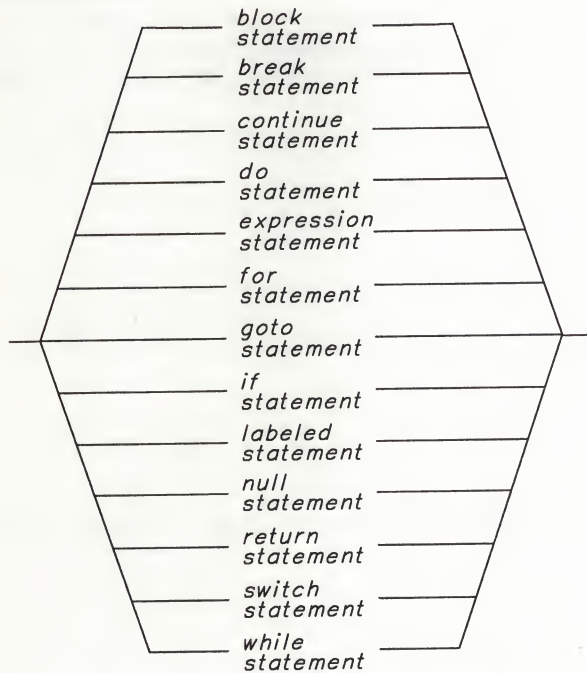
About This Chapter .....	15-3
Block .....	15-4
<b>break</b> .....	15-6
<b>continue</b> .....	15-9
<b>do</b> .....	15-12
Expression .....	15-14
<b>for</b> .....	15-15
<b>goto</b> .....	15-18
<b>if</b> .....	15-20
Labeled .....	15-23
Null .....	15-24
<b>return</b> .....	15-26
<b>switch</b> .....	15-28
<b>while</b> .....	15-34

---

## About This Chapter

This chapter describes the C language statements.

*statement*



---

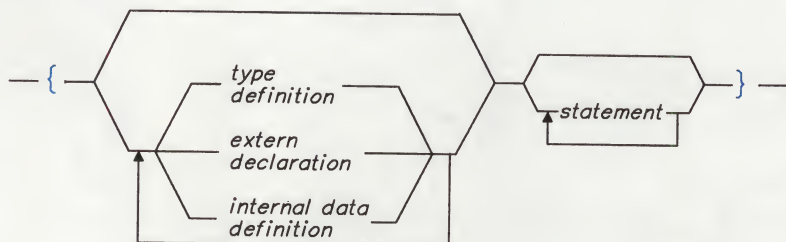
# Block

---

## Description

A **block statement** enables you to group any number of data definitions, declarations, and statements into one statement. A block statement encloses all definitions, declarations, and statements in a single set of braces. You can place a block anywhere a statement is allowed. The block statement has the form:

*block  
statement*



All definitions and declarations occur at the beginning of a block. Statements can follow the definitions and declarations.

## Execution

The compiler treats a block as one compound statement.

If you give a data object inside a block the same identifier as a data object outside that block, the inner object hides the outer object while the block executes. Defining several variables that have the same identifier can make a program difficult to understand and maintain. Therefore, limit such “redefinitions” of identifiers.

If a data object is usable within a block, all nested blocks can use that data object (unless that data object identifier is redefined).

Initialization of an **auto** or **register** variable occurs each time the block executes from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed.

## Examples

The following example shows how the values of data objects change in nested blocks:

```
1  #include <stdio.h>
2
3  main()
4  {
5      int x = 1;                /* Initialize x to 1 */
6      int y = 3;
7
8      if (y > 0)
9      {
10         int x = 2;            /* Initialize x to 2 */
11         printf("second x = %4d\n", x);
12     }
13     printf("first  x = %4d\n", x);
14 }
```

The preceding example produces the following output:

```
second x =    2
first  x =    1
```

Two variables named `x` are defined in `main`. The definition of `x` on line 5 retains storage throughout the execution of `main`. However, since the definition of `x` on line 10 occurs within a nested block, line 11 recognizes `x` as the variable defined on line 10. Line 13 is not part of the nested block. Thus, line 13 recognizes `x` as the variable defined on line 5.

---

# break

---

## Description

A **break statement** contains the word **break** and a semicolon. A **break** statement has the form:

*break*  
*statement*

— **break** — ; —

## Execution

In a looping statement, the **break** statement ends the loop and moves control to the next statement outside the loop.

In a **switch** body, the **break** statement ends the **switch** body and moves control to the next statement outside the **switch** body.

## Restrictions

You can place a **break** statement only in the action part of a looping statement or in the body of a **switch** statement.

## Examples

The following example shows a **break** statement in the action part of a **for** statement. If the *i*th element of the array *string* is equal to '\0', the **break** statement causes the **for** statement to end.

```
for (i = 0; i < 5; i++)  
{  
    if (string[i] == '\0')  
        break;  
    length++;  
}
```

---

The preceding **for** statement is equivalent to the following **for** statement, if string does not contain any embedded NULL characters:

```
for (i = 0; i < 5; i++)
{
    if (string[i] != '\0')
        length++;
}
```

The following example shows a **break** statement in a nested looping statement. The outer loop sequences through an array of pointers to strings. The inner loop examines each character of the string. When the **break** statement executes, the inner loop ends and control returns to the outer loop.

```
/*
** This program counts the characters in the strings that are part of
** an array of pointers to characters. The count stops when one of the
** digits 0 (zero) through 9 is encountered and resumes at the
** beginning of the next string.
*/

#include <stdio.h>

main()
{
    static char *strings[3] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;

    for (i = 0; i < 3; i++)                /* for each string */
                                        /* for each character */
        for (pointer = strings[i]; *pointer != '\0'; ++pointer)
        {
            /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                break;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);
}
```

---

The preceding program produces the following output:

```
letter count = 4
```

The following example is a **switch** statement that contains several **break** statements. Each **break** statement signals the end of a specific clause and ends the execution of the **switch** statement.

```
switch (day)
{
    case 1:
        printf("Monday\n");
        break;

    case 2:
        printf("Tuesday\n");
        break;

    case 3:
        printf("Wednesday\n");
        break;

    case 4:
        printf("Thursday\n");
        break;

    case 5:
        printf("Friday\n");
        break;

    default:
        printf("Invalid weekday.\n");
        break;
}
```

## Related Information

“Looping” on page 3-10.

“**switch**” on page 15-28.

---

## continue

---

### Description

A ***continue statement*** contains the word **continue** and a semicolon. A **continue** statement has the form:

```
continue  
statement  
— continue — ; —
```

### Execution

The **continue** statement ends the execution of the action part of a looping statement and moves control to the condition part of the statement. If the looping statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

### Restrictions

You can place a **continue** statement only in a looping statement.

---

## Examples

The following example shows a **continue** statement in a **for** statement. The **continue** statement causes the system to skip over those elements of the integer array `rates` that have values less than or equal to 1.

```
#include <stdio.h>
main()
{
    int i;
    static float rates[5] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

    printf("Rates over 1.00\n");
    for (i = 0; i < 5; i++)
    {
        if (rates[i] <= 1.00) /* skip rates <= 1.00 */
            continue;
        printf("rate = %.2f\n", rates[i]);
    }
}
```

The preceding program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a **continue** statement in a nested loop. When the inner loop encounters a number in the array `strings`, the **continue** statement ends the inner loop. Execution continues with the re-initialization step of the inner loop.

---

```

/* This program counts the characters in strings that are part
** of an array of pointers to characters. The count excludes
** the digits 0 (zero) through 9.
*/
#include <stdio.h>
main()
{
    static char *strings[3] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < 3; i++) /* for each string          */
        /* for each character */
        for (pointer = strings[i]; *pointer != '\0'; ++pointer)
            /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                continue;
            letter_count++;
    }
    printf("letter count = %d\n", letter_count);
}

```

The preceding program produces the following output:

```
letter count = 5
```

Compare the preceding program with the program on page 15-7. Although the programs are similar, the program on page 15-7 shows the usage of the **break** statement.

## Related Information

“Looping” on page 3-10.

---

# do

---

## Description

A **do statement** contains the word **do** followed by a statement, the word **while**, and an expression in parentheses. A **do** statement has the form:

```
do
statement
— do — statement — while — ( — expression — ) — ; —
```

## Execution

The **do** clause (the statement located between the keywords **do** and **while**) executes before the **while** clause (the expression located after the **while** keyword) is evaluated. Further execution of the **do** statement depends on the value of the **while** clause. If the **while** clause does not evaluate to 0 (zero), the statement executes again. Otherwise, execution of the statement ends.

A **break**, **return**, or **goto** statement can cause the execution of a **do** statement to end, even when the **while** clause does not evaluate to 0 (zero).

## Examples

The following statement prompts the system user to enter a 1 (one) or a 2. If the system user enters a 1 or a 2 the statement ends execution. Otherwise, the statement displays another prompt.

```
do
{
    printf("Enter a 1 or 2.\n");
    scanf("%c", &reply1);
    scanf("%c", &reply2);
    while(reply2 != '\n')
        ;
} while (reply1 != '1' && reply1 != '2');
```

---

## Related Information

“Looping” on page 3-10.

“**break**” on page 15-6.

“**continue**” on page 15-9.

---

## Expression

---

### Description

An **expression statement** contains expression. An expression statement has the form:

*expression*  
*statement*

— *expression* — ; —

### Execution

An expression statement usually assigns a value to a variable or calls a function.

### Examples

```
printf("Account Number: \n");      /* A call to the printf      */
marks = dollars * exch_rate;        /* An assignment to marks    */
(difference < 0) ? ++losses : ++gain; /* A conditional increment   */
switches = flags | BIT_MASK;        /* An assignment to switches */
```

### Related Information

Chapter 3, "Creating Expressions and Statements."  
Chapter 13, "Expressions and Operators."

---

# for

---

## Description

A **for statement** contains the word **for** followed by a list of expressions enclosed in parentheses (the condition part of the statement) and a statement (the action part of the statement). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons. A **for** statement has the form:

*for statement*

— **for** — ( — *expression* — ; — *expression* — ; — *expression* — ) — *statement* —

## Execution

The compiler evaluates the first expression only before the action executes for the first time. You can use this expression to initialize a variable.

The compiler evaluates the second expression before each execution of the action. If this expression evaluates to 0 (zero), the action does not execute and control moves to the next statement following the **for** statement. Otherwise, the action executes.

The compiler evaluates the third expression after each execution of the action. You can use this expression to increment, decrement, or re-initialize a variable.

A **break**, **return**, or **goto** statement can cause the execution of a **for** statement to end, even when the second expression does not evaluate to 0 (zero). If you omit the second expression, you must use a **break**, **return**, or **goto** statement to stop the execution of the **for** statement.

---

## Examples

The following **for** statement prints the value of `count` 20 times. The **for** statement initially sets the value of `count` to 1. After each execution of the action, the statement increments `count`.

```
for (count = 1; count <= 20; count++)
    printf("count = %d\n", count);
```

The following **for** statement does not contain an initialization expression.

```
for (; index > 10; --index)
{
    list[index] = var1 + var2;
    printf("list[index] = %d\n", list[index]);
}
```

The following **for** statement causes the system to execute the action an infinite number of times or until `scanf` receives the letter `e`:

```
for (;;)
{
    scanf("%c", &letter);
    if (letter == '\n')
        continue;
    if (letter == 'e')
        break;
    printf("You entered the letter %c\n", letter);
}
```

The following **for** statement contains multiple initializations and incrementations. The comma operator makes this construction possible.

```
for (i = 0, j = 50; i < 10; ++i, j += 50)
{
    ;
}
```

---

The following example shows a nested **for** statement. The outer statement executes as long as the value of `row` is less than 5. Each time the outer statement executes, the inner statement evaluates `column`. The inner statement executes as long as the value of `column` is less than 3. This example prints the values of an array having the dimensions `[5][3]`:

```
for (row = 0; row < 5; row++)  
    for (column = 0; column < 3; column++)  
        printf("%d\n", table[row][column]);
```

## Related Information

“Looping” on page 3-10.

“**break**” on page 15-6.

“**continue**” on page 15-9.

---

## goto

---

### Description

A **goto statement** consists of the word **goto** followed by an identifier and a semicolon. The identifier must match a statement label in the current function. A **goto** statement has the form:

*goto  
statement*

— **goto** — *identifier* — ; —

### Execution

The **goto** statement moves control to the statement indicated by the identifier.

### Notes

Use the **goto** statement sparingly. Because the **goto** statement can interfere with the normal top-to-bottom sequence of execution, the **goto** makes a program more difficult to read and maintain. Often, a **break** statement, a **continue** statement, or a function call can eliminate the need for a **goto** statement.

---

## Examples

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

```
void display(matrix)
int matrix[3][3];
{
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
                goto out_of_bounds;
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    return;
out_of_bounds: printf("number must be 1 through 6\n");
}
```

## Related Information

“Labeled” on page 15-23.

---

## if

---

### Description

An **if statement** contains the word **if** followed by an expression in parentheses (the condition), a statement, and an optional **else clause**. The else clause contains the word **else** followed by a statement. An if statement has the form:

*if*  
*statement*

— **if** — ( — *expression* — ) — *statement* — { — **else** — *statement* — }

### Execution

The compiler evaluates the condition. If the condition evaluates to a nonzero value, the following statement executes and the system ignores the **else** clause. If the condition evaluates to 0 (zero) and an **else** clause exists, the statement in the **else** clause executes. Otherwise, the next statement executes.

### Notes

The C language does not reserve the word **then**. Thus **then** is implied, but never written, in an **if** statement.

### Examples

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
    grade = 'A';
```

---

The following example displays `number is positive` if the value of `number` is greater or equal to 0 (zero). Otherwise, the example displays `number is negative`.

```
if (number >= 0)
    printf("number is positive\n");
else
    printf("number is negative\n");
```

The following example shows a nested **if** statement:

```
if (paygrade == 7)
    if (level >= 0 && level <= 8)
        salary *= 1.05;
    else
        salary *= 1.04;
else
    salary *= 1.06;
```

The following example shows an **if** statement that does not have an **else** clause. Sometimes omitting an **else** clause within a nested **if** statement is necessary. Because an **else** clause always associates with the closest **if** statement, braces may be necessary to force a particular **else** clause to associate with the correct **if** statement. In this example, omitting the braces would cause the **else** clause to associate with the nested **if** statement.

```
if (gallons > 0)
{
    if (miles > gallons)
        mpg = miles/gallons;
}
else
    mpg = 0;
```

---

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, an action statement executes and the entire **if** statement ends.

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```

## Related Information

“Conditional” on page 3-8.

---

## Labeled

---

### Description

A ***labeled statement*** contains an identifier followed by a colon and a statement. The identifier and colon form the ***label***. A statement can contain more than one label, but each label must contain a different identifier. A labeled statement has the form:

*labeled  
statement*

```
↑ identifier — : — statement —
```

### Execution

A label does not affect the execution of the statement it identifies. A **goto** statement moves control to a labeled statement.

### Examples

```
error: ; /* A labeled null statement. */
init:  units = job(request); /* A labeled expression statement. */
```

### Related Information

“Identifiers” on page 7-6.  
“goto” on page 15-18.

---

# Null

---

## Description

The ***null statement*** consists of a semicolon. A null statement has the form:

```
null  
statement  
———; ———
```

## Execution

The null statement does nothing. You can use a null statement in a looping statement to show a nonexistent action or in a labeled statement to hold the label.

## Examples

The following example initializes the elements of the array `price`:

```
for (i = 0; i < 3; price[i++] = 0)  
    ;
```

---

The following example uses a null statement to hold the label `out_of_bounds:`. When a `matrix` element is out of bounds, the **goto** statement moves control to the labeled statement. The label precedes a null statement because control must move to the end of the loop action and the closing brace of the action is not a statement.

```
void display(matrix)
int matrix[3][3];
{
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
                goto out_of_bounds;
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    out_of_bounds: ;
}
```

---

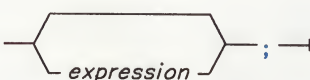
# return

---

## Description

A **return statement** consists of the word **return** followed by an optional expression and a semicolon. A **return** statement has the form:

*return  
statement*

— **return** —  ; —

## Execution

A **return** statement ends the execution of a function and moves control to the calling statement.

A **return** statement is optional. If the system reaches the end of a function without reaching a **return** statement, control passes to the calling statement in the previous function. A value is not returned.

A function can contain multiple **return** statements.

## Value

The C language does not define the value of a **return** statement that does not contain an expression.

A **return** statement that contains an expression has the value of its expression. If the data type of the expression differs from the data type of the expected value, the compiler converts the type of the returned value to the type of the expected value.

---

## Examples

```
return;           /* Returns no value          */
return result;    /* Returns the value of result */
return 1;         /* Returns the value 1          */
return (x * x);   /* Returns the value of x * x    */
```

The following function searches through an array of integers to determine if a match exists for the variable number. If a match exists, the function match returns the value of i. If a match does not exist, the function match returns the value -1 (-one).

```
int match(number, array, n)
int number;
int array[ ];
int n;
{
    int i;

    for (i = 0; i < n; i++)
    {
        if (number == array[i])
            return (i);
    }
    return(-1);
}
```

## Related Information

Chapter 4, "Creating and Using Functions."

Chapter 12, "Functions."

---

# switch

---

## Description

A **switch statement** contains the word **switch** followed by an expression in parentheses and a **switch body**. The expression must have an integral type. A **switch** statement has the form:

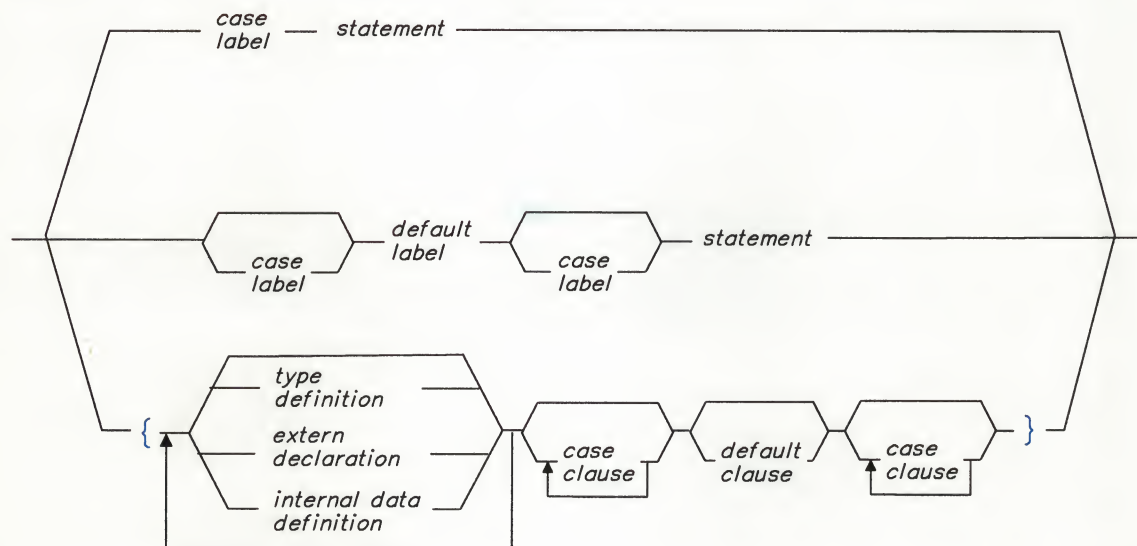
*switch  
statement*

— **switch** — ( — *expression* — ) — *switch  
body* —

A **switch body** can have a simple or complex form. The simple form contains any number of **case** labels mixed with an optional **default** label. The simple form ends with a single statement. Because only the final **case** or **default** label can be followed by a statement, the simple form of the **switch** body is rarely used in C language programs. The **if** statement usually can replace a **switch** statement that has a simple **switch** body.

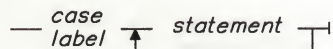
The complex form of a **switch** body is enclosed in braces and can contain definitions, declarations, **case** clauses, and a **default** clause. Each **case** and **default** clause can contain statements. The first two lines of the following diagram show the forms of a simple **switch** body. The last line shows the form of a **complex** body.

*switch*  
*body*



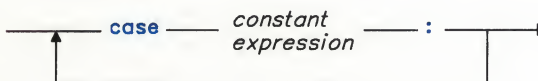
A **case clause** contains a **case** label followed by any number of statements. A **case clause** has the form:

*case*  
*clause*



A **case label** contains the word **case** followed by a constant expression and a colon. Anywhere you can place one **case** label, you can place multiple **case** labels. A **case label** has the form:

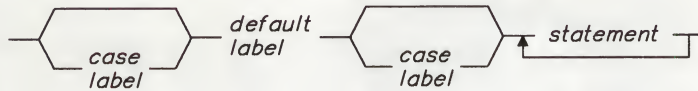
*case*  
*label*



---

A **default clause** contains a **default** label followed by one or more statements. You can place a **case** label on either side of the **default** label. A **default** clause has the form:

*default  
clause*



A **default label** contains the word **default** and a colon. A **switch** statement can have only one **default** label. A **default** label has the form:

*default  
label*

— **default** — : —

## Execution

The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the **switch expression**.

The system compares the value of the **switch** expression with the value of the expression in each **case** label. If the system finds a matching value, control passes to the statement following the **case** label that contains the matching value. If the system does not find a matching value and a **default** label appears anywhere in the **switch** body, control passes to the statement following the **default** label. Otherwise, control passes to the statement following the **switch** body.

If control passes to a statement in the **switch** body, control does not pass from the **switch** body until a **break** statement is encountered or the last statement in the **switch** body is executed.

---

## Restrictions

The **switch** expression and the **case** expressions must be constant expressions having an integral type. All such expressions that do not have type **int** are converted to type **int**.

Each **case** expression must represent a different value.

Only one **default** label can occur in each **switch** statement.

You can place data definitions at the beginning of the **switch** body. However, the compiler does not initialize **auto** and **register** variables at the beginning of a **switch** body.

## Examples

The following **switch** statement contains several **case** clauses and one **default** clause. Each clause contains a function call and a **break** statement. The **break** statements prevent control from passing down through each statement in the **switch** body.

If the **switch** expression evaluated to **'/'**, the switch statement would call the function **divide**. Control would then pass to the statement following the **switch** body.

```
switch (key)
{
    case '+':
        add();
        break;
    case '-':
        subtract();
        break;
    case '*':
        multiply();
        break;
    case '/':
        divide();
        break;
    default:
        printf("invalid key\n");
        break;
}
```

---

The following **switch** statement contains several **case** labels that are not immediately followed by statements:

```
switch (month)
{
    case 12:
    case 1:
    case 2:
        printf("month %d is a winter month\n", month);
        break;

    case 3:
    case 4:
    case 5:
        printf("month %d is a spring month\n", month);
        break;

    case 6:
    case 7:
    case 8:
        printf("month %d is a summer month\n", month);
        break;

    case 9:
    case 10:
    case 11:
        printf("month %d is a fall month\n", month);
        break;

    default:
        printf("not a valid month\n");
        break;
}
```

---

If the expression `month` had the value 3, the system would pass control to the statement:

```
printf("month %d is a spring month\n", month);
```

The **break** statement would pass control to the statement following the **switch** body.

## Related Information

“Conditional” on page 3-8.

“**break**” on page 15-6.

---

# while

---

## Description

A **while statement** contains the word **while** followed by an expression in parentheses (the condition) and a statement (the action). A **while** statement has the form:

*while*  
*statement*

— **while** — ( — *expression* — ) — *statement* —

## Execution

The system evaluates the condition before the action executes. If the condition evaluates to 0 (zero), the action never executes. Otherwise, the action executes. After the action executes, the system re-evaluates the condition. Further execution of the action depends on the value of the condition.

A **break**, **return**, or **goto** statement can cause the execution of a **while** statement to end, even when the condition does not evaluate to 0 (zero).

---

## Examples

In the following program, `item[index]` triples each time the value of the expression `--index` is greater than or equal to 0 (zero). When `--index` evaluates to a negative number, the **while** statement ends.

```
#define MAX_INDEX 5
#include <stdio.h>
main()
{
    static int item[ ] = { 12, 55, 62, 85, 102 };
    int index = MAX_INDEX;

    while (--index >= 0)
    {
        item[index] *= 3;
        printf("item[%d] = %d\n", index, item[index]);
    }
}
```

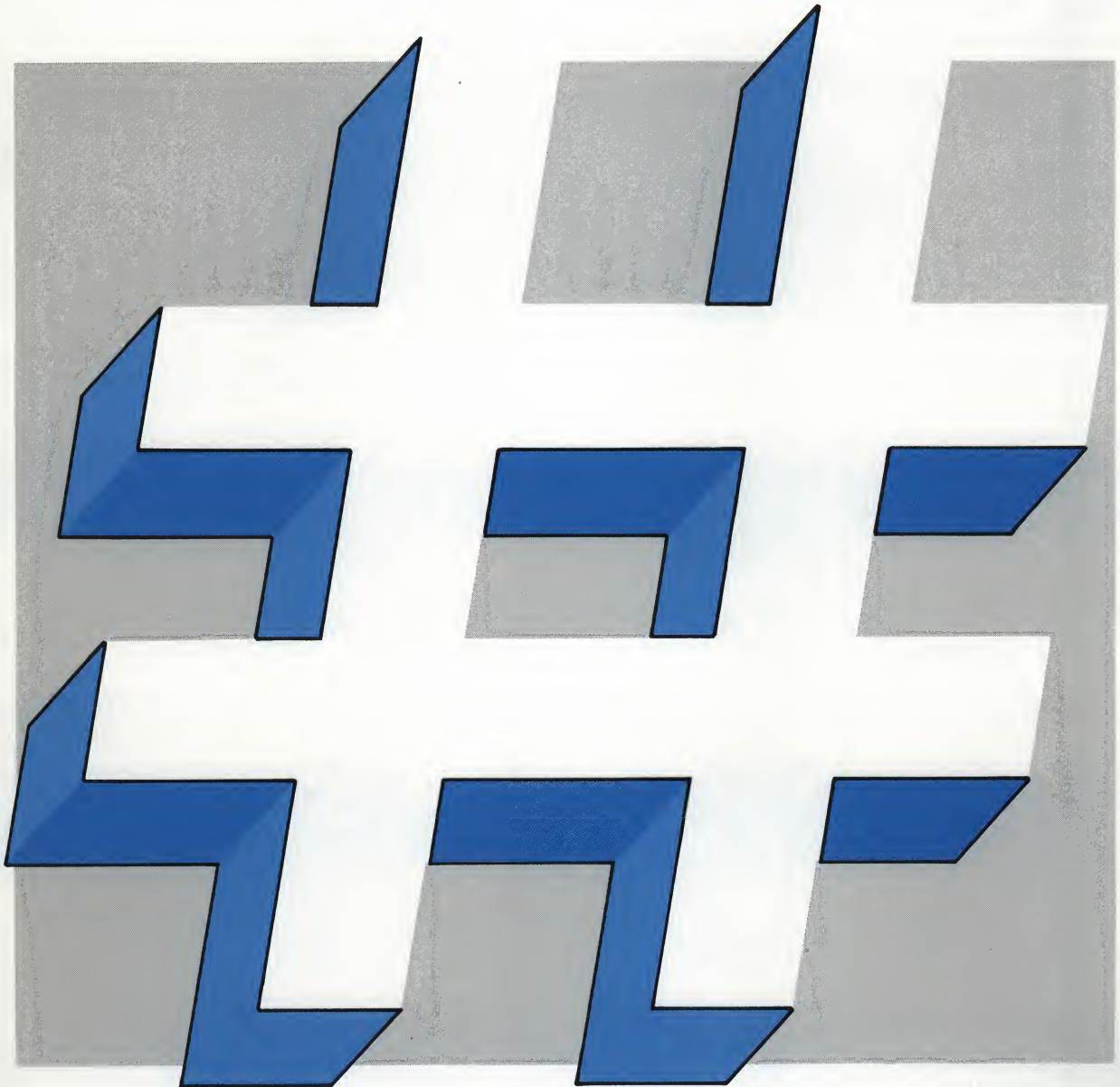
## Related Information

“Looping” on page 3-10.  
“**break**” on page 15-6.  
“**continue**” on page 15-9.



---

## Chapter 16. Preprocessor Statements



---

## CONTENTS

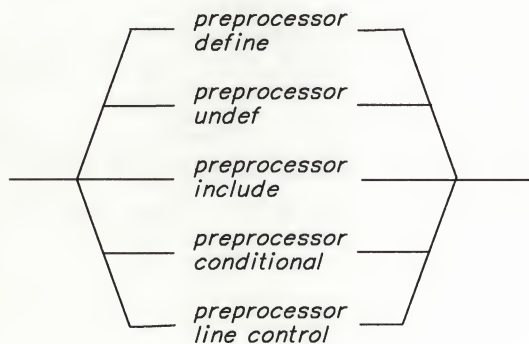
About This Chapter .....	16-3
Preprocessor Statement Format .....	16-4
<b>define</b> .....	16-5
<b>undef</b> .....	16-9
<b>include</b> .....	16-10
Conditional Compilation .....	16-13
<b>if</b> .....	16-15
<b>ifdef</b> .....	16-15
<b>ifndef</b> .....	16-16
<b>line</b> Control .....	16-18

---

## About This Chapter

This chapter describes the C preprocessor statements. The preprocessor, rather than the compiler, interprets preprocessor statements. The ***preprocessor*** is a program that prepares C language programs for compilation. The **cc** command automatically sends programs through the preprocessor, then sends the output of the preprocessor through the compiler. The preprocessor recognizes the following types of statements:

*preprocessor  
statement*



Preprocessor statements enable you to:

- Replace identifiers or strings in the current file with specified code
- Embed files within the current file
- Conditionally compile sections of the current file
- Change the line number of the next line of code and change the file name of the current file.

---

## Preprocessor Statement Format

---

Preprocessor statements begin with the # character followed by a preprocessor keyword. The # character must appear as the first character on a line. The IBM RT PC C Language preprocessor requires that the # character be located in the first column of a line. Only space and tab characters can separate the # and the preprocessor keyword. The remainder of the line can be filled with arguments to the preprocessor, C language comments, and white space.

When the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ as a continuation marker. The preprocessor ignores the \ (and the following new-line character) and interprets the following line as a continuation of the current preprocessor line.

Preprocessor statements can appear any place in a program. They cannot, however, appear on the same line as C language code that is not part of a preprocessor statement.

The effect of a preprocessor statement lasts until the end of the source file in which the statement appears.

---

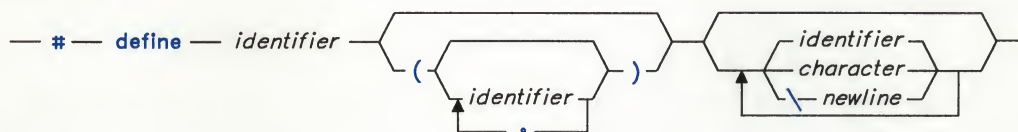
# define

---

## Description

A **preprocessor define statement** causes the preprocessor to replace an identifier or macro with specified code. A preprocessor **define** statement has the form:

*preprocessor  
define*



The **define** statement can contain a simple macro definition or a complex macro definition.

## Simple Macro Definition

A **simple macro definition** replaces a single identifier with another identifier or with a string of characters and identifiers. The following simple definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000`:

```
#define COUNT 1000
```

This definition would cause the preprocessor to change the following statement (if the statement appeared after the previous definition and in the same file as the definition):

```
int array[COUNT];
```

In the output of the preprocessor, the preceding statement would appear as:

```
int array[1000];
```

The following definition references the previously defined identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

---

## Complex Macro Definition

A **complex macro definition** receives parameters from a macro call, embeds these parameters in some replacement code, and substitutes the replacement code for the macro call. A complex definition is an identifier followed by a parenthesized parameter list and the replacement code. White space cannot separate the identifier (which is the name of the macro) and the parameter list. A comma must separate each parameter.

A **macro call**, like a function call, is an identifier followed by a parenthesized list of arguments. Unlike a function call, white space cannot separate the identifier and the argument list. A comma must separate each argument.

The following line defines the macro SUM as having two parameters a and b and the replacement code (a + b):

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appeared after the previous definition and in the same file as the definition):

```
c = SUM(x,y);  
c = d * SUM(x,y);
```

In the output of the preprocessor, the preceding statement would appear as:

```
c = (x + y);  
c = d * (x + y);
```

## Notes

A macro call must have the same number of arguments as the corresponding macro definition has parameters.

In the macro call argument list, commas that appear as character constants, in string constants or surrounded by parentheses, do not separate arguments.

A definition is not required to specify replacement code. The following definition removes all instances of the word `static` from subsequent lines in the current file:

```
#define static
```

You can change the definition of a defined identifier or macro with a second **preprocessor define** statement or with a **preprocessor undef** statement (see “**undef**” on page 16-9).

Within the text of the program, the preprocessor does not scan character constants or string constants for macro calls.

---

## Examples

The following program contains two macro definitions and a macro call that references both of the defined macros:

```
#define SQR(s) ( (s) * (s) )
#define PRNT(a,b) printf("value 1 = %d\n", a); \
printf("value 2 = %d\n", b)

main()
{
    int x = 2;
    int y = 3;

    PRNT(SQR(x),y);
}
```

After being interpreted by the preprocessor, the preceding program appears as follows:

```
# 1 "macro.c"

main()
{
    int x = 2;
    int y = 3;

    printf("value 1 = %d\n", ( (x) * (x) )); printf("value 2 = %d\n", y);
}
```

In the preceding example, the preprocessor inserted the line:

```
# 1 "macro.c"
```

The preprocessor inserted this line, which indicates the line number 1 (one) and the name of the file in which the program was stored (macro.c), so that any line number references to the preprocessed code would match the line numbers in the original source code.

Execution of this program produces the following output:

```
value 1 = 4
value 2 = 3
```

---

## Related Information

“Compiling and Linking” on page 5-4.  
“**undef**” on page 16-9.

---

# undef

---

## Description

A **preprocessor *undef* statement** causes the preprocessor to end the scope of a preprocessor definition. A preprocessor **undef** statement has the form:

*preprocessor*  
*undef*

— # — **undef** — *identifier* —

## Examples

The following statements define BUFFER and SQR:

```
#define BUFFER 512  
#define SQR(x) (x) * (x)
```

The following statements nullify the preceding definitions:

```
#undef BUFFER  
#undef SQR
```

Occurrence of the identifiers BUFFER and SQR that appear following these **undef** statements are not substituted for the previously defined code.

## Related Information

“**define**” on page 16-5.

---

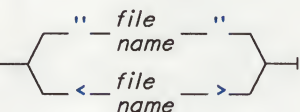
# include

---

## Description

A **preprocessor include statement** causes the preprocessor to replace the statement with the contents of the specified file. A preprocessor **include** statement has the form:

*preprocessor  
include*

— # — include —  —

The diagram shows a horizontal line with a bracket on the left and a vertical line on the right. Between them, there are two paths. The top path goes up and then right, with a double quote character at the start and end, and the text "file name" in the middle. The bottom path goes down and then right, with an angle bracket character at the start and end, and the text "file name" in the middle.

If the file name is enclosed in double quotation marks, the preprocessor searches the directory that contains the source file, then a standard or specified sequence of directories until it finds the specified file. For example:

```
#include "lib/payroll.h"
```

If the file name is enclosed in the characters < and >, the preprocessor searches only the standard or specified directories for the specified file. For example:

```
#include <stdio.h>
```

---

## Usage

If you have a number of definitions that several files use, you can place all these definitions in one file and **include** that file in each file that must know the definitions. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of definitions:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "/u/david/defs.h"
```

You can embed the definitions that appear in `defs.h` with the following statement:

```
#include "defs.h"
```

The preprocessor would look for the file `defs.h` first in the directory that contains the source file. If not found there, the preprocessor would search a sequence of specified or standard places.

If the file name begins with the `/` character, the preprocessor searches only the specified directory for the file. For example:

```
#include "/u/david/defs.h"
```

---

## Notes

The C language does not define how you can specify a sequence of directories for the preprocessor to search. The command `cc`, however, recognizes the flag `-Idirectory`, which enables you to specify a directory for the preprocessor to search before searching the standard directories. Assume the file `pgm.c` contains the following statement:

```
#include "in-file"
```

If `pgm.c` were compiled using the following command:

```
cc -Imelanie/include pgm.c
```

The preprocessor would search for the file `in-file` in the following directories:

1. The directory that contains the file `pgm.c`
2. The directory `melanie/include`
3. The standard sequence of directories.

If instead, the file `pgm.c` contained the statement:

```
#include <in-file>
```

The preprocessor would search for the file `in-file` in the following directories:

1. The directory of `melanie/include`
2. The standard sequence of directories.

## Related Information

“Compiling and Linking” on page 5-4.

---

# Conditional Compilation

---

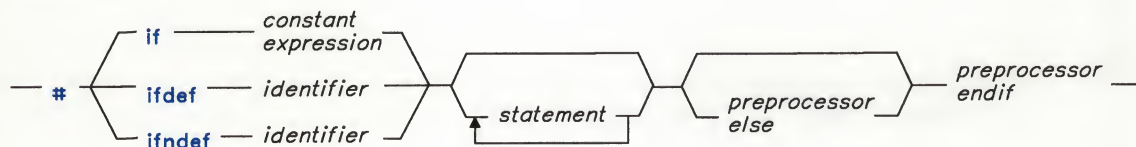
## Description

A **preprocessor conditional compilation statement** causes the preprocessor to insert specified code in the file depending on how a specified condition evaluates. A preprocessor conditional compilation statement spans several lines:

1. The condition specification line
2. Lines containing code that the preprocessor inserts in the program if the condition evaluates to a nonzero value (optional)
3. The **else** line (optional)
4. Lines containing code that the preprocessor inserts in the program if the condition evaluates to 0 (zero) (optional)
5. The preprocessor **endif** statement.

A preprocessor conditional compilation statement has the form:

*preprocessor  
conditional*



A preprocessor conditional compilation statement can have one of three types of conditions: **if**, **ifdef**, and **ifndef**.

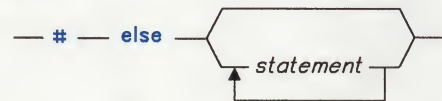
The following table describes the usages of each:

Condition	Description
<b>if</b>	Inserts the code that immediately follows the condition if the condition evaluates to a nonzero value.
<b>ifdef</b>	Inserts the code that immediately follows the condition if the identifier specified in the condition is defined.
<b>ifndef</b>	Inserts the code that immediately follows the condition if the identifier specified in the condition is not defined.

### Figure 16-1. Preprocessor Conditional Compilation Directives

If the condition evaluates to 0 (zero), or false, and the conditional compilation statement contains a preprocessor **else** statement, the preprocessor inserts the lines that appear between the preprocessor **else** statement and the preprocessor **endif** statement. Otherwise, the preprocessor deletes these lines. The preprocessor **else** statement has the form:

```
preprocessor
else
```



The preprocessor **endif** statement ends the conditional compilation statement. The preprocessor **endif** statement has the form:

```
preprocessor
endif
```

```
— # — endif —
```

You can nest preprocessor conditional statements.

---

## if

The **if** keyword must be followed by a constant expression. The constant expression cannot contain a **sizeof** expression or an enumeration constant. For example:

```
#if TEST >= 1
    printf("i = %d\n", i);
    printf("array[i] = %d\n", array[i]);
#endif
```

The constant expression can contain the keyword **defined**. This keyword can be used only with the preprocessor keyword **if**. The expression:

**defined** *identifier*

**OR**

**defined**(*identifier*)

evaluates to 1 if the *identifier* is defined in the preprocessor, otherwise to 0 (zero). For example:

```
#if defined(TEST1) || defined(TEST2)
```

## ifdef

An identifier must follow the **ifdef** keyword. The following example defines **SIZEOF\_INT** to be 32 if **I80286** is defined for the preprocessor. Otherwise, **SIZEOF\_INT** is defined to be 16.

```
#ifdef I80286
#   define SIZEOF_INT 32
#else
#   define SIZEOF_INT 16
#endif
```

---

## ifndef

An identifier must follow the **ifndef** keyword. The following example defines `SIZEOF_INT` to be 16 if `I80286` is not defined for the preprocessor. Otherwise, `SIZEOF_INT` is defined to be 32.

```
#ifndef I80286
#   define SIZEOF_INT 16
#else
#   define SIZEOF_INT 32
#endif
```

## Notes

The command `cc` recognizes the flag `-Didentifier`, which enables you to specify at compile time an identifier for the preprocessor to define. For example, the following command defines the identifier `I80286` in the file `pgm.c`:

```
cc -DI80386 pgm.c
```

## Examples

The following example shows how you can nest preprocessor conditional compilation statements:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#       define MAX_PHASE 2
#   else
#       define MAX_PHASE 8
#   endif
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#endif
```

---

The following program contains preprocessor conditional compilation statements:

```
main()
{
    static int array[ ] = { 1, 2, 3, 4, 5 };
    int i;

    for (i = 0; i <= 4; i++)
    {
        array[i] *= 2;

#ifdef TEST
        printf("i = %d\n", i);
        printf("array[i] = %d\n", array[i]);
#endif
    }
}
```

---

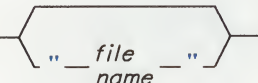
## line Control

---

### Description

A **preprocessor line control statement** causes the compiler to view the line number of the next source line as the specified number. A preprocessor **line** statement has the form:

*preprocessor  
line control*

— # — line — *decimal  
constant* —  —

A file name specification enclosed in quotes can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the file specified by the preceding **line** control statement. If a **line** control statement does not precede the current statement, the compiler views the line as part of the current source file.

### Notes

The compiler recognizes the identifiers **--LINE--** and **--FILE--**. **--LINE--** evaluates to the current line number. The identifier **--FILE--** evaluates to the current file name. Thus, the following statement prints an error message that contains the current line number and file name:

```
printf("Error on line %d in file %s.\n", --LINE--, --FILE--);
```

The preprocessor and other programs may produce line control statements (other than those specified in the file). For example, if the first line of a file is an **include** statement, the preprocessor inserts the specified file and a **line** control statement that sets the number of the line that follows the **included** code to 2.

---

## Examples

You can use **line** control statements to make the compiler provide more meaningful error messages. The following program uses **line** control statements to give each function an easily recognizable line number:

```
#include <stdio.h>
main()
{
    func_1();
    func_2();
}

#line 100
func_1()
{
    printf("Func_1 - the current line number is %d\n", __LINE__);
}

#line 200
func_2()
{
    printf("Func_2 - the current line number is %d\n", __LINE__);
}
```

The preceding program produces the following output:

```
Func_1   The current line number is 102.
Func_2   The current line number is 202.
```



---

## Appendix A. IBM RT PC C Language

This appendix describes some characteristics of the IBM RT PC C Language compiler. These characteristics can vary on other C language compilers.

### Data Types

The following table lists the C language data types and their size characteristics on the IBM RT PC C Language compiler:

Data Type	Bits	Range	Precision
<b>char</b>	8	'\0' to '\377' (0 to 255)	
<b>unsigned char</b>	8	'\0' to '\377' (0 to 255)	
<b>short</b>	16	-32,768 to 32,767	
<b>unsigned short</b>	16	0 to 65,535	
<b>int</b>	32	-2,147,483,648 to 2,147,483,647	
<b>unsigned int</b>	32	0 to 4,294,967,295	
<b>long</b>	32	-2,147,483,648 to 2,147,483,647	
<b>unsigned long</b>	32	0 to 4,294,967,295	
<b>float</b>	32	approx. $-3.37e+38$ to $3.37e+38$	7 digits
<b>double</b>	64	approx. $-1.67e+308$ to $1.67e+308$	15 digits
<b>long double</b>	64	approx. $-1.67e+308$ to $1.67e+308$	15 digits
<b>pointer</b>	32		

Figure A-1. IBM RT PC C Language Data Type Size Specifications

The **char** type is **unsigned**, and the **int** type is signed.

---

## Floating-Point Representation

The IBM RT PC C Language compiler supports the floating-point format defined by the ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. When you write a floating-point value to a display or a printer, one of the following special values may appear in place of the floating-point number:

Value	Meaning
QNaN	Quiet NaN (not a number)
SNaN	Signalling NaN
+INF or -INF	$\pm \infty$
+0 or -0	$\pm 0$

**Figure A-2. Special Floating-Point Values**

The values QNaN and SNaN may be preceded by a sign. The sign, however, does not affect the interpretation of the value. ANSI/IEEE standard 754-1985 defines these special floating-point values.

## Data Layout

Within a word, the low-order byte is at the highest address.

Data objects are aligned as follows:

Data Type	Alignment
char	On byte boundaries
short	On halfword boundaries
int, long, float, double, pointer	On fullword boundaries

**Figure A-3. IBM RT PC C Language Alignment of Data**

Structure and union variables are aligned according to the maximum alignment of their members.

---

## Bit Fields

Fields are assigned from left to right. You can define bit fields as having type **int** or **unsigned int**. However, bit fields always receive type **unsigned int**. Each group of bit fields occupies a sequence of 32-bit words. Bit fields do not cross word boundaries. You can initialize bit fields.

## Reserved Words

Along with the usual reserved words, the IBM RT PC C Language compiler reserves **asm**, **fortran**, and **volatile**. You can use:

`asm(assembler code)`

**OR**

`asm("assembler code")`

to embed assembler code in a C language program. Embedding assembler code can disturb the contents of registers that the compiler uses. Therefore, take great care when using the keyword **asm**.

although the keyword **fortran** is reserved, the IBM RT PC C Language compiler does not associate any meaning with **fortran**.

See "volatile Attribute" on page 8-9 for information about the keyword **volatile**.

The IBM RT PC C Language compiler also reserves external identifiers beginning with `_C_` prefix. See *AIX Operating System Technical Reference* for information about any AIX Operating System functions that have names beginning with `_C_`.

## Identifiers

The compiler uses at least the first 64 characters in an identifier (internal or external) as significant characters. Uppercase and lowercase letters are distinguished. The compiler appends the character `_` (underscore) to the beginning of external identifiers.

---

## Register Variables

The compiler provides seven registers for variables having integral or pointer types and four registers for variables having type **double**.

## Internal Variables

Each function can address at most 32,767 bytes of internal data. This data includes:

- All internal variables (except register variables)
- All temporary variables produced by the compiler
- Up to 76 bytes for linkage
- Enough space for the size of the maximum argument list built by the function.

The compiler does not place additional restrictions on the number of internal variables in a function.

## Structures

The compiler enables functions to return a structure. When a function having the declared return type structure is called, the calling function passes a pointer to a temporary memory area which will contain the result. The pointer becomes the first argument to the function, all other arguments to the function are displaced accordingly. The calling function must correctly declare a function that returns a structure, even if the result of the call is ignored.

## Right Shift

The compiler performs an arithmetic right shift when right shifting a signed variable. The arithmetic right shift fills vacated bits with a copy of the sign bit.

## Constants

A character constant can contain one to four characters, but can occupy no more than four bytes of storage. Thus, you can place four 1-byte characters, or only two 2-byte characters, in a character constant. The IBM RT PC C Language compiler treats character constants as having type **unsigned int**.

A float constant can end with the letter f or F. The f or F causes the constant to have type **float**.

The compiler does not limit the following:

- The number of digits in an integer constant
- The length of a string constant.

---

Truncation can occur, however, if the value of an integer constant or string constant cannot be represented in the system wordsize.

## Conversions

The IBM RT PC C Language compiler performs single precision operations when all operands have type **float**. Thus, **float** objects are not automatically converted to **double** objects before these operations. An exception to this occurs in function calls. All **float** arguments are converted to type **double** before their values are passed to another function.

## The Memory Model

**malloc** allocates from within the data segment. The segment registers are used as follows:

Segment Register	Usage
0	kernel
1	text
2	data
3	stack
4 through 13	available through the shared memory operations

## The cc Command and the Preprocessor

The following items are not limited by the IBM RT PC C Language compiler:

- The number of arguments allowed on a compiler command line
- The number of command line arguments available to a compiled program
- The number of macro arguments
- The maximum nesting of macro arguments
- The maximum nesting of conditionals
- The number of include files
- The maximum nesting of include files.

---

## Parameter Passing

You can view the set of parameters to a function as an array of words, each parameter beginning in a new word. The first four words are passed in registers. Additional words are passed on the stack. Values are passed as follows:

- An **int** object is passed in a single word or register, fullword aligned.
- **char**, **short**, and **long** objects and pointers to data objects are treated as **int** objects and are right justified in a single word or in a register.
- A pointer to a function is passed like a normal pointer. The address passed is that of the function constant pool; its first word contains the function entry point.
- A **double** object is passed in two successive words which may not be doubleword aligned. One word may be in a register, and the other word may be in the stack.
- A **float** object is converted to a **double** object.
- A structure is aligned to a fullword and occupies as many words as the structure needs. If the last word is only partially filled, the contents of the last word are left-justified.

Declaring the first four words of parameters as having the storage class **register** can increase program efficiency. Taking the address of a parameter can decrease program efficiency.

## Other Limitations

The IBM RT PC C Language compiler contains the following additional limitations:

- The amount of heap space is 256 megabytes less the size of the data and stack segments in the object module.
- The maximum number of open files is 250. For portable programs, however, you should limit open files to 20.
- The maximum number of array dimensions is 13. For portability, limit the number of array dimensions to four.
- The limit to the size of a single stack frame is 32K bytes. A stack frame contains **auto** and temporary variables, arguments to routines called from within the frame, and a save area of no more than 128 bytes.

The IBM RT PC C Language compiler does not limit the following:

- Number of names in an object module
- Depth of struct nesting
- Depth of struct initialization
- Number of expressions in a block
- Number of expressions in a statement
- Number of statements in a block

- 
- Number of statements in a function
  - Level of block nesting
  - Level of loop nesting
  - Complexity limits of expressions.



---

## Appendix B. Portability Considerations

You can write C language programs that compile and run on various systems. The following list contains some suggestions for making your programs portable:

- Avoid writing system-dependent code. If you must write system-dependent code, keep it in one place and note that place in a comment at the beginning of your program.
- Avoid using compiler enhancements to the C language.
- Avoid using preprocessor statements that are not described in Chapter 16, "Preprocessor Statements."
- Avoid using system functions that are not part of the C Language Standard Library.
- Make internal identifiers distinct in the first eight characters and external identifiers distinct in the first six characters.
- Do not rely on the uppercase and lowercase characteristics of a letter to make identifiers distinct. For example, some systems may interpret the identifiers `account` and `Account` as names for the same data object.
- Be aware of any limits that are imposed by the compilers that run your programs.



---

## Appendix C. Common Errors in C Programming

This appendix describes some errors that both experienced and inexperienced C language programmers make. The following list describes syntax errors that cause compiler messages:

- A missing ; (semicolon) at the end of a statement. For example:

```
main()
{
    statement;
    statement;
    statement    /* This statement requires a semicolon.    */
}
```

- A reference to the *n*th element in an array that contains *n* objects. For example:

```
char code[9]; /* Elements range from code[0] to code[8]. */
code[9] = 0;  /* The element code[9] does not exist.    */
```

- An attempt to initialize an **auto** aggregate. For example:

```
auto message[2] = {"No match. ", "Match found. "};
```

The following list describes errors that may not produce compiler messages. These errors, however, affect the way the compiler interprets the code:

- Usage of the assignment operator = where the equality operator == is meant.

```
total = 5;    /* Assigns 5 to total.    */
total == 5;   /* Yields 1 (one) if the value of total
               /* is 5; yields 0 (zero) otherwise.    */
```

- Omission of the & symbol before a non-pointer variable in a **scanf** function call. For example:

```
int x;
scanf("%d", x);
```

- 
- Precedence confusion. For example:

```
if (x == y() != 2)      /* This if expression always evaluates */
                        /* to 0 (zero). */
if (x == (y() != 2) )   /* This if expression may evaluate to */
                        /* 0 (zero) or 1 (one). */
```

- Expectation of a called function to change a value of an object in the calling function. If you pass the address of a value (using & in the calling function), the called function can change that value by pointing to its contents (using \*). See “Calling Functions and Passing Values” on page 12-12.
- Declaration of an array without dimensions. This provides lexical scope only; no space is allocated. For example:

```
char names[ ];
```

---

## Appendix D. Advanced Example Program

The following file `source1.c` contains part of the advanced example program:

```
/* source1.c */

/*
** This program creates a linked list of strings from an array of
** pointers to strings, sequences through the linked list, and writes
** the strings to an output file.
*/

#include <stdio.h>

struct record
{
    char *name;
    struct record *next;
} *top_of_list = NULL;

main()
{
    extern void insert(), create_file();
    static char *word-- = {
        "crush", "apple", "crab",
        "jump", "and", NULL
    };

    int i;

    for (i = 0; word[i] != NULL; i++)
        insert(word[i]);
    create_file();
}
```

---

The following file `source2.c` contains the remainder of the advanced example program:

```
/* source2.c */

#include <stdio.h>

extern struct record
{
    char *name;
    struct record *next;
} *top_of_list;

/*
** This function creates a linked list of words linked in ascending
** order.
*/
void insert(newword)
char *newword;
{
    struct record *p, *prev_p, *new_p;

    for (p = top_of_list; p != NULL; prev_p = p, p = p->next)
    {
        if (strcmp(newword, p->name) <= 0)
            break;
    }

    new_p = (struct record *) malloc(sizeof(*new_p));

    new_p->name = newword;
    new_p->next = p;

    if (p == top_of_list)
        top_of_list = new_p;
    else
        prev_p->next = new_p;
}
```

---

```

/*
** This function opens an output file for writing, sequences through
** a linked list, and outputs a string to the output file using the
** fprintf standard library function.
*/
void create_file()
{
    FILE *out_file , *fopen();
    static char out_name[10];
    struct record *rec_p;

    printf("Enter name of output file.\n");
    scanf("%s", out_name);

    out_file = fopen(out_name, "w");
    if (out_file == NULL)
        printf("Cannot open file %s for writing.\n", out_name);

    for (rec_p = top_of_list; rec_p != NULL; rec_p = rec_p->next)
        fprintf(out_file, "%s\n", rec_p->name);
}

```

The following command compiles the advanced example program and produces the executable file a.out:

```
cc source1.c source2.c
```

Interaction with the program can produce the following session:

```

Input:  a.out
Output: Enter name of output file.
Input:  test
Output: cat test
        and
        apple
        crab
        crush
        jump

```



---

## Appendix E. Utilities for C Programmers

### **cb**

Reformats a source program into a consistent, indented format. “**cb**” on page 6-8 further describes the **cb** program. See also *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

### **cflow**

Generates a flow diagram of a program. See *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

### **cxref**

Generates a cross reference listing of a program. See *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

### **ed**

Helps you enter a source program into a file. **ed** is a line editor. See *AIX Operating System Commands Reference*.

### **INed<sup>1</sup> Program**

Helps you enter a source program into a file. **INed** is a full screen editor. See *IBM RT PC INed*.

### **lint**

Checks a program for syntax, type conversion, and portability problems. Generates a listing of these problems. “**lint**” on page 6-5 further describes the **lint** program. See also *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

### **make**

Builds programs from several source modules. The **make** program compiles only those modules that you change. See *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

---

<sup>1</sup> Registered trademark of INTERACTIVE Systems Corporation

---

### **Source Code Control System**

Maintains separate versions of a program without storing separate copies of each version. See *AIX Operating System Programming Tools and Interfaces* or *AIX Operating System Commands Reference*.

### **Symbolic Debugger (sdb)**

Helps you to find and fix program logic problems. See also *AIX Operating System Programming Tools and Interfaces* and *AIX Operating System Commands Reference*.

---

## Appendix F. RT PC Character Codes

This appendix lists the decimal, octal, hexadecimal, and character representations for ASCII standard characters and for other characters supported on the IBM RT PC.

The list shows only characters available on Code Page P0 in the code page format for the RT PC. Other characters are available on two other code pages, Code Page P1 and Code Page P2.

For information on how to access characters on Code Page P1 and Code Page P2, see *AIX Operating System Technical Reference. Managing the AIX Operating System* lists the characters that are available through Code Page P1 and Code Page P2.

Decimal Value	Octal Value	Hex Value	Character Value	Decimal Value	Octal Value	Hex Value	Character Value
000	000	00	NUL	043	053	2B	+
001	001	01	SOH	044	054	2C	,
002	002	02	STX	045	055	2D	—
003	003	03	ETX	046	056	2E	.
004	004	04	EOT	047	057	2F	/
005	005	05	ENQ	048	060	30	0
006	006	06	ACK	049	061	31	1
007	007	07	BEL	050	062	32	2
008	010	08	BS	051	063	33	3
009	011	09	HT	052	064	34	4
010	012	0A	LF	053	065	35	5
011	013	0B	VT	054	066	36	6
012	014	0C	FF	055	067	37	7
013	015	0D	CR	056	070	38	8
014	016	0E	SO	057	071	39	9
015	017	0F	SI	058	072	3A	:
016	020	10	DLE	059	073	3B	;
017	021	11	DC1	060	074	3C	<
018	022	12	DC2	061	075	3D	=
019	023	13	DC3	062	076	3E	>
020	024	14	DC4	063	077	3F	?
021	025	15	NAK	064	100	40	@
022	026	16	SYN	065	101	41	A
023	027	17	ETB	066	102	42	B
024	030	18	CAN	067	103	43	C
025	031	19	EM	068	104	44	D
026	032	1A	SUB	069	105	45	E
027	033	1B	ESC	070	106	46	F
028	034	1C	SS4	071	107	47	G
029	035	1D	SS3	072	110	48	H
030	036	1E	SS2	073	111	49	I
031	037	1F	SS1	074	112	4A	J
032	040	20	BLANK (SPACE)	075	113	4B	K
033	041	21	!	076	114	4C	L
034	042	22	''	077	115	4D	M
035	043	23	#	078	116	4E	N
036	044	24	\$	079	117	4F	O
037	045	25	%	080	118	50	P
038	046	26	&	081	121	51	Q
039	047	27	'	082	122	52	R
040	048	28	(	083	123	53	S
041	051	29	)	084	124	54	T
042	052	2A	*	085	125	55	U

Decimal Value	Octal Value	Hex Value	Character Value	Decimal Value	Octal Value	Hex Value	Character Value
086	126	56	V	129	201	81	Ù
087	127	57	W	130	202	82	Ú
088	130	58	X	131	203	83	Û
089	131	59	Y	132	204	84	Ü
090	132	5A	Z	133	205	85	Ý
091	133	5B	[	134	206	86	Þ
092	134	5C	\	135	207	87	ÿ
093	135	5D	]	136	210	88	é
094	136	5E	^	137	211	89	ê
095	137	5F	_	138	212	8A	ë
096	140	60	`	139	213	8B	ì
097	141	61	a	140	214	8C	í
098	142	62	b	141	215	8D	î
099	143	63	c	142	216	8E	ï
100	144	64	d	143	217	8F	ÿ
101	145	65	e	144	220	90	É
102	146	66	f	145	221	91	æ
103	147	67	g	146	222	92	Æ
104	150	68	h	147	223	93	Ò
105	151	69	i	148	224	94	ó
106	152	6A	j	149	225	95	Ô
107	153	6B	k	150	226	96	Ù
108	154	6C	l	151	227	97	Ú
109	155	6D	m	152	230	98	ÿ
110	156	6E	n	153	231	99	ö
111	157	6F	o	154	232	9A	ü
112	160	70	p	155	233	9B	ø
113	161	71	q	156	234	9C	£
114	162	72	r	157	235	9D	Ø
115	163	73	s	158	236	9E	×
116	164	74	t	159	237	9F	ƒ
117	165	75	u	160	240	A0	á
118	166	76	v	161	241	A1	í
119	167	77	w	162	242	A2	ó
120	170	78	x	163	243	A3	Û
121	171	79	y	164	244	A4	ü
122	172	7A	z	165	245	A5	ÿ
123	173	7B	{	166	246	A6	ß
124	174	7C		167	247	A7	Ð
125	175	7D	}	168	250	A8	ˆ
126	176	7E	~	169	251	A9	˜
127	177	7F	Δ	170	252	AA	Ł
128	178	80	Ç	171	253	AB	½

Decimal Value	Octal Value	Hex Value	Character Value	Decimal Value	Octal Value	Hex Value	Character Value
172	254	AC	¼	214	326	D6	í
173	255	AD	í	215	327	D7	î
174	256	AE	«	216	330	D8	ï
175	257	AF	»	217	331	D9	
176	260	B0		218	332	DA	
177	261	B1		219	333	DB	■
178	262	B2		220	334	DC	■
179	263	B3		221	335	DD	·
180	264	B4	┐	222	336	DE	
181	265	B5	A	223	337	DF	■
182	266	B6	À	224	340	E0	o
183	267	B7	À	225	341	E1	ö
184	270	B8		226	342	E2	
185	271	B9		227	343	E3	
186	272	BA		228	344	E4	
187	273	BB		229	345	E5	
188	274	BC		230	346	E6	
189	275	BD		231	347	E7	
190	276	BE		232	350	E8	
191	277	BF		233	351	E9	
192	300	C0	┐	234	352	EA	
193	301	C1	┐	235	353	EB	
194	302	C2	┐	236	354	EC	
195	303	C3	┐	237	355	ED	
196	304	C4	┐	238	356	EE	
197	305	C5	┐	239	357	EF	
198	306	C6		240	360	F0	
199	307	C7		241	361	F1	
200	310	C8		242	362	F2	
201	311	C9		243	363	F3	
202	312	CA		244	364	F4	
203	313	CB		245	365	F5	
204	314	CC		246	366	F6	
205	315	CD		247	367	F7	
206	316	CE		248	370	F8	
207	317	CF		249	371	F9	
208	320	D0		250	372	FA	
209	321	D1		251	373	FB	
210	322	D2		252	374	FC	
211	323	D3		253	375	FD	
212	324	D4		254	376	FE	
213	325	D5		255	377	FF	BLANK 'FF'

---

## Figures

2-1.	Constants .....	2-4
2-2.	Storage Classes .....	2-7
2-3.	Type Specifiers .....	2-9
3-1.	Operators .....	3-4
3-2.	Preprocessor Keywords .....	3-14
4-1.	<b>printf</b> Conversion Codes .....	4-10
4-2.	<b>printf</b> Conversion Modifiers .....	4-11
4-3.	<b>scanf</b> Conversion Codes .....	4-15
4-4.	<b>scanf</b> Conversion Modifiers .....	4-16
5-1.	<b>cc</b> Flags .....	5-5
6-1.	<b>lint</b> Flags .....	6-6
6-2.	<b>cb</b> Flags .....	6-8
7-1.	The C Language Reserved Words .....	7-8
7-2.	Additional Words Reserved by Some Compilers .....	7-8
8-1.	Example Declarators .....	8-10
9-1.	Escape sequences .....	9-11
13-1.	Operator Precedence and Associativity .....	13-4
14-1.	Usual Unary Conversions .....	14-4
14-2.	Arithmetic Widening Conversions .....	14-5
14-3.	Assignment Conversions .....	14-7
16-1.	Preprocessor Conditional Compilation Directives .....	16-14
A-1.	IBM RT PC C Language Data Type Size Specifications .....	A-1
A-2.	Special Floating-Point Values .....	A-2
A-3.	IBM RT PC C Language Alignment of Data .....	A-2



---

## Glossary

**access.** To obtain data from or put data in storage.

**address.** A name, label, or number identifying a location in storage.

**aggregate.** An array, a structure, or a union.

**allocate.** To assign resource.

**American National Standard Code for Information Interchange (ASCII).** The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

**a.out.** An output file produced by default from the **cc** command. This file is executable.

**argument.** In a function call, an expression that represents a value the calling function passes to the function specified in the call.

**arithmetic object.** An integral object or objects having the type **float** or **double**. The IBM RT PC C Language compiler also recognizes objects having the type **long double** as arithmetic objects.

**array.** A variable that contains an ordered group of data objects. All objects in an array have the same data type.

**ASCII.** See *American National Standard Code for Information Interchange*.

**assembler language.** A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond

directly to the storage and registers of the machine.

**assignment conversion.** A change to the form of the right operand that makes the right operand have the same data type as the left operand.

**assignment expression.** An operation that stores the value of the right operand in the storage location specified by the left operand.

**associativity.** The order for grouping operands with an operator (either left-to-right or right-to-left).

**binary.** (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

**binary expression.** An operation containing two operands and one operator.

**bit field.** A member of a structure or union that contains one or more named bits.

**block statement.** Any number of data definitions, declarations, and statements that appear between the symbols { and }. The compiler reads a block statement as a single C language statement.

**boundary alignment.** The position in main storage of a fixed-length field (such as halfword or doubleword) on an integral boundary for that unit of information. For example, a word boundary is a storage address evenly divisible by four.

**break statement.** A C language control statement that contains the word **break** and a semicolon.

**byte.** The C language does not define the size of a byte. However, most C language compilers (including the IBM RT PC C Language compiler) consider the size of a **char** object as the size of a byte.

**C language.** A general-purpose high-level programming language.

**C language statement.** A C language statement contains zero or more expressions. All C language statements, except block statements, end with a **;** symbol. A block statement begins with a **{** symbol, ends with a **}** symbol, and contains any number of statements.

**case clause.** In a **switch** statement, a **case** label followed by any number of statements.

**case label.** The word **case** followed by a constant expression and a colon.

**cast.** An expression that converts the value of the operand to a specified data type (the operator).

**char specifier.** The words **char** or **unsigned char** which describe the type of data a variable represents.

**character.** A letter, a digit, or another symbol.

**character constant.** A character or an escape sequence enclosed in single quotation marks. Some compilers allow more than one character or escape sequence in a character constant. The IBM RT PC C Language compiler allows you to place one to four characters in a character constant. The character constant, however, can occupy no more than four bytes of storage. Thus, although you can place four 1-byte characters in a character constant, you can place only two 2-byte characters in a character constant.

**character set.** A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

**character variable.** A data object whose value can be changed during program execution and whose data type is **char** or **unsigned char**.

**code.** Instructions for the computer system.

**comma expression.** An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. If the left operand produces a value, the compiler discards this value.

**command.** A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

**comment.** A comment contains text that the compiler ignores. Comments begin with the **/\*** characters, end with the **\*/** characters, and span any number of lines. Comments cannot be nested.

**compilation time.** The time during which a source program is translated from a high-level language (such as C language) into a machine language.

**compile.** The computer actions required to transform a source file into an executable object file.

**compiler.** A program that translates instructions written in a high-level programming language (such as C language) into a machine language.

**condition.** An expression that can be evaluated to a value of either true or false when a program is compiled or run.

**conditional compilation statement.** A preprocessor statement that causes the preprocessor to insert specified code in the file depending on how a specified condition evaluates.

**conditional expression.** A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value 0 (zero).

**conditional statement.** A C language statement that executes if a specified expression evaluates to nonzero.

**constant.** A data object with a value that does not change. Contrast with *variable*.

**constant expression.** An expression having a value that is determined during compilation and that cannot be changed during execution.

**continue statement.** A C language control statement that contains the word **continue** and a semicolon.

**control statement.** A C language statement that changes the normal path of execution.

**conversion.** A change in the form of a value. For example, when you add values having different data types, the compiler converts both values to the same form before adding the values.

**conversion code.** In a **printf** function call, a specification of the type of the value, as the value is to be printed (in octal format, for example).

**conversion modifier.** In a **printf** function call, a specification of how a value is to be printed (left justified, for example).

**conversion specification.** In a **printf** function call, a specification of how the system is to place the value of zero or more format parameters in the output stream. Each conversion specification contains a % symbol. The % is followed by conversion modifiers and a conversion code.

**data definition.** A data definition describes a data object, and reserves storage. A data

definition can also provide an initial value. Definitions appear outside a function or at the beginning of a block statement.

**data type.** A category that identifies the mathematical qualities and internal representation of data.

**debug.** To detect, locate, and correct mistakes in a program.

**decimal.** Pertaining to a system of numbers to the base 10; decimal digits range from 0 through 9.

**decimal constant.** A number containing any digits 0 through 9 that does not begin with 0 (zero).

**declaration.** A description that makes a defined object available to a function or a block.

**declarator.** An identifier and optional symbols that further describe the data type. (The IBM RT PC C Language compiler allows you to include the optional keyword **volatile** in the declarator portion of a variable definition.)

**default.** A value that is used when no alternative is specified by the programmer.

**default clause.** In a **switch** statement, a **default** label followed by one or more statements.

**default initialization.** The initial value of the data object if an initializer is not specified. **extern** and **static** variables receive 0 (zero) as their default initial values. **auto** and **register** variables receive undefined default initial values.

**default label.** The word **default** followed by a colon.

**define statement.** A preprocessor statement that causes the preprocessor to replace an identifier or macro call with specified code.

**definition.** In programming languages, a data type description or a data object description that reserves storage and, sometimes, provides an initial value.

**digit.** Any of the numerals from 0 through 9.

**directory.** A type of file containing the names and controlling information for other files or other directories.

**do statement.** A C language looping statement that contains the word **do** followed by a statement (the action), the word **while**, and an expression in parentheses (the condition).

**double precision.** Pertaining to the use of two computer words to represent a number in accordance with the required precision.

**EBCDIC.** See *extended binary-coded decimal interchange code*.

**element.** A data object in an array.

**else clause.** The part of a C language **if** statement that contains the word **else** followed by a statement. The **else** clause provides an action that is executed when the **if** condition evaluates to nonzero (false).

**enumeration constant.** An identifier (that has an associated integer value) defined in an enumerator. You can use an enumeration constant anywhere an integer constant is allowed.

**enumeration data type.** A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

**enumeration tag.** The identifier that names an enumeration data type.

**enumerator.** An enumeration constant and its associated value.

**escape sequence.** A representation of a character. An escape sequence contains the **\** symbol followed by one of the characters: **b**, **f**,

**n**, **r**, **t**, **v**, **'**, **"**, or **\** or followed by one to three octal digits.

**exponent.** A number, indicating the power to which another number (the base) is to be raised.

**expression.** A representation of a value. For example, variables and constants appearing alone or in combination with operators.

**expression statement.** An expression that ends with a **;** (semicolon). You can use an expression statement to assign the value of an expression to a variable or to call a function.

**extended binary-coded decimal interchange code (EBCDIC).** A set of 256 eight-bit characters.

**external data definition.** A description of a variable appearing outside a function that causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition.

**file.** A collection of related data that is stored and retrieved by an assigned name.

**file name.** The name used by a program to identify a file.

**flag.** A modifier that appears on a command line with the command name that defines the action of the command. Flags are often preceded by a dash.

**float constant.** A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an **e** or **E**, an optional sign (**+** or **-**), and one or more digits **0** through **9**. The IBM RT PC C Language compiler allows you to place an optional **f** or **F** at the end of a floating-point constant. (Some compilers may generate error messages when they encounter an **f** or **F** at the end of a floating-point constant.)

**for statement.** A C language looping statement that contains the word **for** followed

---

by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

**function.** A named group of statements that can be called and evaluated and that can return a value to the calling statement.

**function call.** An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of arguments.

**function declarator.** The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

**function definition.** The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

**function header.** The part of a function declarator that names the function and lists the function parameters.

**goto statement.** A C language control statement that contains the word **goto** followed by an identifier and a semicolon. The identifier must match a statement label in the current function.

**hexadecimal.** Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and uppercase or lowercase A (ten) through F (fifteen).

**hexadecimal constant.** The characters **0x** or **0X** (zeroX) followed by any digits 0 through 9

and uppercase or lowercase letters A through F.

**high-order.** Most significant; leftmost. For example, bit 0 in a register.

**identifier.** A name that you use to reference a data object. An identifier can contain letters, digits, and underscores. A digit, however, cannot appear as the first character in an identifier.

**if statement.** A C language conditional statement that contains the word **if** followed by an expression in parentheses (the condition), a statement (the action), and an optional **else** clause (the alternate action).

**include statement.** A preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file.

**initialize.** To set the starting value of a data object.

**initializer.** The assignment operator followed by an expression (or multiple expressions, for aggregate variables).

**input.** Data to be processed.

**int specifier.** The words **int**, **short**, **short int**, **long**, **long int**, **unsigned**, **unsigned int**, **unsigned short**, **unsigned short int**, **unsigned long**, or **unsigned long int** which describe the type of data a variable represents.

**integer.** A positive or negative whole number or zero.

**integer constant.** A decimal, octal, or hexadecimal constant.

**integral object.** A character object, an object having an enumeration type, or an object having the type **short**, **int**, **long**, **unsigned short**, **unsigned int**, or **unsigned long**.

**internal data definition.** A description of a variable appearing at the beginning of a block that causes the system to allocate storage for

that variable and makes that variable accessible to the current block.

**keyword.** A predefined word.

**label.** One or more identifiers followed by a colon that can precede a statement.

**labeled statement.** A C language statement that contains one or more identifiers followed by a colon and a statement.

**letter.** An uppercase or lowercase character from the set A through Z.

**library.** A collection of functions, calls, subroutines, or other data.

**line control statement.** A preprocessor statement that causes the compiler to view the line number of the next source line as the specified number.

**linefeed.** An ASCII character that causes an output device to move forward one line.

**linkage editor.** A program that resolves cross-references between separately assembled object modules, then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linkage editor simply makes it relocatable.

**long constant.** An integer constant followed by the letter l (el) or L.

**loop.** A statement performed repeatedly until an ending condition is reached.

**looping statement.** A C language statement that executes any number of times, depending on the value of a specified expression.

**low-order.** Least significant; rightmost. For example, in a 32 bit register (0-31), bit 31 is the low-order bit.

**lvalue.** An expression that represents a data object that can be both examined and altered.

**machine language.** A language that can be used directly by a computer without intermediate processing.

**macro call.** An identifier followed by a parenthesized list of arguments that the preprocessor replaces with the replacement code located in a preprocessor define statement.

**main function.** A function that has the identifier **main**. Each program must have exactly one function named **main**. This function begins and ends program execution.

**matrix.** An array arranged in rows and columns.

**member.** A data object in a structure or a union.

**memory.** Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers. See *storage*.

**message.** A response from the system to inform the operator of a condition which may affect further processing of a current program.

**nest.** To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one block (the nested subroutine) within another block (the nesting block).

**new-line character.** A control character that causes the print or display position to move to the first position on the next line.

**null.** Having no value, containing nothing.

**null character (NUL).** The character hex 00, used to represent the absence of a printed or displayed character.

**null statement.** A C language statement that consists of a semicolon.

**object code.** Machine-executable instructions, usually generated by a compiler from source

---

code written in a higher level language (such as C language). For programs that must be linked, object code consists of relocatable machine code.

**octal.** A base eight numbering system.

**octal constant.** The digit 0 (zero) followed by any digits 0 through 7.

**operand.** An identifier, a constant, or an expression that is grouped with an operator.

**operating system.** Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

**operation.** A specific action (such as add, multiply, shift) that the computer performs when requested.

**operator.** A symbol (such as +, -, \*) that represents an operation (in this case, addition, subtraction, multiplication).

**output.** The result of processing data.

**overflow.** A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

**pad.** To fill unused positions in a field with data, usually zeros, ones, or blanks.

**parameter.** A value that a function receives.

**parameter declaration.** A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

**pointer.** A variable that holds the address of a data object.

**precedence.** The priority system for grouping different types of operators with their operands.

**precision.** A measure of the ability to distinguish between nearly equal values. See *single precision* and *double precision*.

**preprocessor.** A program that examines the source program for preprocessor statements which are then executed, resulting in the alteration of the source program.

**preprocessor statement.** A statement that begins with the symbol # and contains instructions that the preprocessor interprets.

**primary expression.** An identifier, a parenthesized expression, a function call, an array element specification, or a structure or union member specification.

**program.** One or more files containing a set of instructions conforming to a particular programming language syntax.

**prompt.** A displayed request for information or user action.

**register.** A storage area, in a computer, capable of storing a specified amount of data such as a bit or an address.

**reserved word.** A word that is defined in a programming language for a special purpose, and that must not appear as a user-declared identifier.

**return statement.** A C language control statement that contains the word **return** followed by an optional expression and a semicolon.

**run.** To cause a program, utility, or other machine function to be performed.

**scalar.** An arithmetic object, or a pointer to an object of any type.

**scope.** That part of a source program in which a variable can communicate its value.

**single precision.** Pertaining to the use of one computer word to represent a number in accordance with the required precision.

**sign balancing.** A conversion that makes both operands have the same data type (signed or unsigned). If one operand has an **unsigned**

---

type, the compiler converts the other operand to that **unsigned** type. Otherwise, both operands remain signed.

**source program.** A set of instructions written in a programming language, that must be translated to machine language before the program can be run.

**statement.** An instruction that ends with the character **;** or several instructions that are surrounded by the characters **{** and **}**.

**storage.** The location of saved information.

**storage class specifier.** A storage class keyword.

**string constant.** Zero or more characters enclosed in double quotation marks.

**structure.** A variable that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

**structure tag.** The identifier that names a structure data type.

**subscript.** One or more expressions, each enclosed in brackets, that follow an array name. A subscript references an element in an array.

**subscript declarator.** In an array definition or declaration, the bracketed expressions following the array name. A subscript declarator specifies the number elements in an array dimension.

**switch expression.** The expression that is located between the word **switch** and the body of a **switch** statement.

**syntax.** The rules for the construction of a command or a program.

**system.** The computer and its associated devices and programs.

**truncate.** To shorten a value to a specified length.

**type balancing.** A conversion that makes all operands have the same size data type. If both of the operands do not have the same size data type, the compiler converts the value of the operand having the smaller type to a value having the larger type.

**type class.** A category of related data types. The C language type classes are: aggregate, scalar, arithmetic, integral, and character.

**type definition.** A definition of a synonym for a data type.

**type specifier.** A name of a data type.

**unary expression.** An expression that contains one operand.

**undef statement.** A preprocessor statement that causes the preprocessor to end the scope of a preprocessor definition.

**union.** A variable that can hold any one of several data types, but only one data type at a time.

**union tag.** The identifier that names a union data type.

**utility.** A program that performs a common service.

**valid.** Allowed.

**value.** The contents of a storage location.

**variable.** A data object with a value that can be changed while the program executes.

**volatile attribute.** The keyword **volatile** located in a definition, a declaration, or a cast. **volatile** causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object.

**while statement.** A C language looping statement that contains the word **while** followed by an expression in parentheses (the condition) and a statement (the action).

---

**white space.** Space characters, tab characters, new-line characters, and comments.

**widening.** An expansion of the size of a value (for example, **short** to **int**) by padding bits located to the left of the value with a copy of the sign bit.

**word.** A contiguous series of 32 bits (four bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

**zero suppression.** The substitution of blanks for leading zeros in a number. For example, 00057 becomes 57 when using zero suppression.



## Special Characters

- . dot operator 13-15
- .c file 5-4
- .o file 5-4
- < less than operator 13-26
- << left-shift operator 13-25
- <<= assignment operator 13-34
- <= less than or equal to operator 13-26
- () operators
  - for calling functions 13-12
  - for grouping expressions 13-12
- + addition operator 13-24
- ++ increment operator 13-17
- += assignment 13-34
- arithmetic negation operator 13-18
- subtraction operator 13-24
- decrement operator 13-17
- > arrow operator 13-15
- = assignment operator 13-34
- & address operator 13-19
- & bitwise operator 13-28
- && logical AND operator 13-29
- &= assignment operator 13-34
- ! logical negation operator 13-19
- != not equal to operator 13-27
- \* indirection operator 13-20
- \* multiplication operator 13-23
- \*= assignment operator 13-34
- ^ bitwise exclusive OR operator 13-28
- = assignment operator 13-34
- / division operator 13-23
- /= assignment operator 13-34
- , comma operator 13-35
- % remainder operator 13-23

- %= assignment operator 13-34
- > greater than operator 13-26
- >> right-shift operator 13-25, A-4
- >>= assignment operator 13-34
- >= greater than or equal to operator 13-26
- ? : conditional operators 13-31
- # preprocessor statement character 3-14, 16-4
- = simple assignment operator 13-33
- == equal to operator 13-27
- ~ bitwise negation operator 13-19
- [ ] array subscript operators 13-14
- ! bitwise inclusive OR operator 13-29
- != assignment operator 13-34
- || logical OR operator 13-30

## A

- a.out 5-4
- additive operators
  - addition + 13-24
  - subtraction - 13-24
- address operator & 13-19
- addressable space
  - internal variables A-4
  - register variables A-4
- aggregate types 13-3
- alignment of data A-2
- AND operator (bitwise) & 13-28
- AND operator (logical) && 13-29
- argc argument count 12-4
- arguments in a function call 4-6, 12-12
- argv argument vector 12-4
- arithmetic conversions, usual
  - sign balancing 14-6
  - type balancing 14-6
  - widening 14-5

---

- arithmetic negation operator - 13-18
- arithmetic types 13-3
- arrays
  - array subscripting 13-14
  - description of 11-8
- arrow operator -> 13-15
- ASCII character codes 9-10, F-1
- asm A-3
- assembler 5-4, A-3
- assignment conversions 14-7
- assignment expressions 13-33
  - complex 13-34
  - simple 13-33
- associativity of operators 13-4
- auto storage class 8-14

## B

- backslash escape sequence \\ 9-10
- backspace escape sequence \b 9-10
- binary expression 13-22
- bit field 11-31, 11-35, A-3
- bitwise negation operator ~ 13-19
- bitwise operators
  - AND & 13-28
  - exclusive OR ^ 13-28
  - inclusive OR | 13-29
  - left-shift << 13-25
  - right-shift >> 13-25
- block statement 3-6, 15-4
- brackets [ ] 13-14
- break statement 3-11, 15-6
- byte 13-21

## C

- C standard library
  - printf function 4-8
  - scanf function 4-14
- call, function 4-6, 12-12
- calls, function 13-12

- carriage return escape sequence \r 9-10
- case clause 15-28, 15-29
- case label 15-29
- cast operator 13-20
- cb program re-formatter 6-8
- cc 16-3, 16-12, 16-16
- cc command 5-4
- char 10-4, A-1
- char constant 9-4
- char specifier 10-4
- character set, RT PC F-1
- character string 9-21
- character, definition of X-4
- characters
  - character constant 9-4
  - data types
    - char 10-4, A-1
    - unsigned char 10-4, A-1
  - escape sequence 9-10
  - string constant 9-21
- comma operator , 13-35
- commands
  - cb 6-8
  - cc 5-4, 16-3, 16-12, 16-16
  - lint 6-5
- comments 7-4
- comments, as white space 1-6, 16-4
- compiler control lines
  - See preprocessor statements
- compiling a program 5-4
- compound statement 3-6, 15-4
- conditional compilation 16-13
  - if preprocessor statement 16-15
  - ifdef preprocessor statement 16-15
  - ifndef preprocessor statement 16-16
- conditional expression ? : 13-31
- conditional statements 3-8
  - if 15-20
  - switch 15-28
- constant expression 13-9
- constants 2-4, 9-3, A-4
  - character 9-4
  - enumeration 9-8
  - escape sequence 9-10
  - float 9-12

- integer 9-17
  - decimal 9-6
  - hexadecimal 9-15
  - long 9-18
  - octal 9-19
- string 9-21
- continue statement 3-11, 15-9
- control statements 3-12
  - break 15-6
  - continue 15-9
  - goto 15-18
  - return 15-26
- conversion code 4-8, 4-10, 4-14, 4-15
- conversion modifier 4-8, 4-11, 4-14, 4-16
- conversion specifications 4-8
- conversions 14-3
  - assignment 14-7
  - cast operator 13-20
  - explicit 14-8
    - expansion 14-8, 14-9
    - pointer 14-8, 14-9
    - reduction 14-8
    - void 14-8, 14-10
    - volatile 14-8, 14-10
  - usual arithmetic 14-5
  - usual unary 14-4

## D

- data definitions
  - See also declarations
  - constant 2-4, 9-3
  - definition of 1-4, 2-3
  - function 4-4
  - variable 2-6
    - external 8-6
    - internal 8-4
- data layout A-2
- data types
  - array 11-8
  - character 10-4
  - enumeration 11-17
  - enumerations 11-17
  - floating-point 10-6

- functions 12-5
- IBM RT PC C Language compiler sizes
  - of A-1
- integer 10-8
- pointer 11-22
- structure 11-30
- union 11-39
- void 10-10, 14-10
- debugging a program
  - cb program re-formatter 6-8
  - error messages 6-4
  - lint program verifier 6-5
- decimal constant 9-6
- declaration, function 12-10
- declarations
  - See also data definitions
  - function 12-10
  - parameter 4-6, 12-7
- declarators 2-6, 2-9, 8-8
  - array 11-8
  - character 10-4
  - floating-point 10-6
  - integer 10-9
  - pointer 11-22
  - structure 11-32
  - union 11-40
- decrement operator -- 13-17
- default clause 15-28, 15-30
- default label 15-30
- define preprocessor statement 5-5, 16-5
- defined, preprocessor keyword 16-15
- definition, macro 16-5
- definitions
  - See also declarations
  - constant 2-4, 9-3
  - definition of 1-4, 2-3
  - function 4-4
  - variable 2-6
    - external 8-6
    - internal 8-4
- diagrams
  - See syntax diagrams
- digit X-6
- division operator / 13-23
- do statement 3-11, 15-12

- dot operator . 13-15
- double 10-6, A-1
- double precision
  - constants 2-5, 9-12
  - definition of X-6
  - storage allocated A-1
  - variables 2-8, 10-6
- double quotation escape sequence \" 9-10

## E

- element, definition of X-6
- else clause 15-20
- else, preprocessor keyword 16-13
- end of string 9-21
- endif, preprocessor line 16-13
- enum 11-17
- enum constant 9-8
- enum specifier 11-17
- enumerations
  - enum data types 11-17
  - enumeration constant 9-8
- enumerator 11-17
- envp environment pointer 12-4
- equal to operator == 13-27
- equality operators
  - See also relational operators
  - equal to == 13-27
  - not equal to != 13-27
- errors
  - common errors C-1
  - messages 6-4, 16-19
- escape character \ 9-10
- escape sequence 9-10
- evaluation, expression 13-4
- exclusive OR operator (bitwise) ^ 13-28
- executable file 5-4, 5-6, 5-7
- expansion conversions 14-8, 14-9
  - floating-point 14-9
  - integral to floating-point 14-9
  - signed arithmetic 14-9
  - unsigned arithmetic 14-9
- exponent 9-12

- expression statement 15-14
- expressions 3-4, 13-3
  - assignment 13-33
  - binary 13-22
  - comma 13-35
  - conditional 13-31
  - constant 13-9
  - evaluation of 13-4
  - lvalue 13-7
  - primary 13-11
  - unary 13-16
- extern declaration 8-19
- extern storage class 8-19
- external data definitions 8-6
  - extern 8-19
  - static 8-27
- external identifier 7-6, A-3

## F

- field, bit 11-31, 11-35
- FILE identifier 16-18
- file inclusion 16-10
- files, number allowed open A-6
- float 10-6, A-1
- float constant 9-12
- float specifier 10-6
- float types
  - double 10-6, A-1
  - float 10-6, A-1
  - long double 10-6, A-1
- floating-point conversions 14-9
- floating-point representation A-2
- floating-point to integral conversions 14-8
- for statement 3-11, 15-15
- form feed escape sequence \f 9-10
- format parameter 4-8, 4-14
- format, program 1-6
- fortran A-3
- function declarator 12-6
- function header 12-6
- functions
  - body 12-9

---

- calling functions 4-6, 13-12
- declarations 12-10
- declarator 12-6
- definition 4-4
- definitions 12-5
- main 4-4, 12-4
- parameter 12-12
- parameter declaration 4-6, 12-7
- return statements 15-26
- void 12-11

## G

- global variables 8-19
- goto statement 3-13, 15-18
- greater than operator > 13-26
- greater than or equal to operator >= 13-26

## H

- heap space A-6
- hexadecimal constant 9-15
- horizontal tab escape sequence \t 9-10

## I

- identifiers
  - attributes of 2-9, 7-6
  - IBM RT PC C Language restrictions A-3
- if preprocessor statement 16-15
- if statement 3-8, 15-20
- ifdef preprocessor statement 16-15
- ifndef preprocessor statement 16-16
- include preprocessor statement 5-5, 16-10
- inclusive OR operator (bitwise) | 13-29

- increment operator ++ 13-17
- indentation of code 1-6, 16-4
- indirection operator \* 13-20
- initial expression 8-12
- initializers 2-6, 2-10, 8-12
  - array 11-10
  - character 10-4
  - floating 10-6
  - integer 10-9
  - pointer 11-23
  - structure 11-32
- input 4-14
- int 10-8, A-1
- int constant 9-17
- int specifier 10-8
- integers
  - data types
    - int 10-8, A-1
    - long 10-8, A-1
    - short 10-8, A-1
    - unsigned 10-8
    - unsigned int 10-8, A-1
    - unsigned long 10-8, A-1
    - unsigned short 10-8, A-1
  - float constant 9-12
  - integer constants 9-17
    - decimal 9-6
    - hexadecimal 9-15
    - long 9-18
    - octal 9-19
- integral reduction conversions 14-8
- integral to floating-point conversion 14-8
- integral to floating-point conversions 14-9
- integral to pointer conversions 14-9
- integral types 13-3
- internal data definitions 8-4
  - auto 8-14
  - register 8-24
  - static 8-27
- internal identifier 7-6
- internal variables A-4

---

**K**

## keywords

- additional IBM RT PC C Language A-3
- list of 7-8

**L**

labeled statement 15-23

layout of data A-2

left-shift operator &lt;&lt; 13-25

less than operator &lt; 13-26

less than or equal to operator &lt;= 13-26

letter, definition of X-8

library, C standard

- printf function 4-8

- scanf function 4-14

line continuation character \

- escape sequence, as an 9-10

- preprocessor statements, in 16-4

- string constants, in 1-6, 9-22

line control preprocessor statement 16-18

line feed escape sequence \r 9-10, 9-22

LINE identifier 16-18

linefeed, definition of X-8

linking source files 5-4

lint program verifier 6-5

local variables 8-4

logical AND operator &amp;&amp; 13-29

logical negation operator ! 13-19

logical OR operator || 13-30

long 10-8, A-1

long constant 9-18

long double 10-6, A-1

looping statements 3-10

- do 15-12

- for 15-15

- while 15-34

lvalue 13-7

**M**

macro call 16-6

macro definition 16-5, 16-6

main function 4-4, 12-4

member 11-31, 11-39

memory model A-5

modulo operator % 13-23

multiplicative operators

- division / 13-23

- multiplication \* 13-23

- remainder % 13-23

**N**

## names

- attributes of 2-9, 7-6

- IBM RT PC C Language restrictions A-3

negation operators

- arithmetic - 13-18

- bitwise ~ 13-19

- logical ! 13-19

new-line character

- escape sequence \n 9-10, 9-22

- white space, as 1-6, 16-4

not equal to operator != 13-27

NUL character \0 9-21

NUL escape sequence \0 9-10

null pointer 11-24, 14-7

null statement 15-24

**O**

object file 5-4

octal constant 9-19

one's complement operator ~ 13-19

open files A-6

operand, definition of 3-4

operators 3-4

- additive
  - addition operator + 13-24
  - subtraction - 13-24
- assignment 13-33
  - <<= 13-34
  - += 13-34
  - &= 13-34
  - \*= 13-34
  - ^= 13-34
  - /= 13-34
  - %= 13-34
  - >>= 13-34
  - = 13-33, 13-34
  - = 13-34
- binary 13-22
- bitwise AND & 13-28
- bitwise exclusive OR ^ 13-28
- bitwise inclusive OR | 13-29
- bitwise shift
  - left-shift << 13-25
  - right-shift >> 13-25, A-4
- comma , 13-35
- conditional ? : 13-31
- equality operators 13-27
  - equal to == 13-27
  - not equal to != 13-27
- logical AND && 13-29
- logical OR || 13-30
- multiplicative
  - division / 13-23
  - multiplication \* 13-23
  - remainder % 13-23
- precedence and associativity 13-4
- primary 13-11
  - array subscripting [ ] 13-14
  - arrow -> 13-15
  - dot . 13-15
  - parentheses ( ) 13-12
- relational operators 13-26
  - greater than > 13-26
  - greater than or equal to >= 13-26
  - less than < 13-26

- less than or equal to <= 13-26
- unary 13-16
  - address operator & 13-19
  - arithmetic negation operator - 13-18
  - bitwise negation operator ~ 13-19
  - cast operator 13-20
  - decrement operator -- 13-17
  - increment operator ++ 13-17
  - indirection operator \* 13-20
  - logical negation operator ! 13-19
  - sizeof operator 13-21
- optimizer 5-6
- OR operator (logical) || 13-30
- output 4-8
- overflow, definition of X-9

## P

- parameter declaration 4-6, 12-7
- parameter passing 12-12, A-6
- parentheses
  - for calling functions 13-12
  - for grouping expressions 13-12
- passing a value 12-12
- passing an address 12-12
- passing by value 4-7
- pointer conversions 14-8, 14-9
  - integral to pointer 14-9
  - pointer to pointer 14-9
- pointer parameter 4-14
- pointers
  - description of 11-22
  - size specification A-1
  - structure and union members, to 13-15
- portability considerations B-1
- precedence of operators 13-4
- precision X-9
  - See also double precision
  - See also single precision
- preprocessor 3-14, 5-4, 16-3
- preprocessor statements 3-14, 16-3
  - conditional compilation 16-13
  - define 5-5, 16-5

- else 16-14
- endif 16-14
- format of 16-4
- include 5-5, 16-10
- line control 16-18
- undef 16-9
- primary expression 13-11
- printf function 4-8
- program verifier, lint 6-5

## R

- recursion
  - function calls 13-14
- reduction conversions 14-8
  - floating-point to integral 14-8
  - integral 14-8
  - integral to floating-point 14-8
- register storage class 8-24
  - IBM RT PC C Language restrictions A-4
- registers
  - asm's affect on A-3
- relational operators
  - See also equality operators
  - greater than > 13-26
  - greater than or equal to >= 13-26
  - less than < 13-26
  - less than or equal to <= 13-26
- remainder operator % 13-23
- reserved words
  - additional IBM RT PC C Language A-3
  - list of 7-8
- return statement 3-13, 15-26
- return value, as declared 12-10
- right-shift operator >> 13-25, A-4
- running a program 5-7

## S

- scalar types 13-3
- scanf function 4-14
- segment registers A-5
- shift operators << and >> 13-25
- short 10-8, A-1
- sign balancing 14-6
- signed arithmetic expansion conversions 14-9
- simple assignment operator = 13-33
- single precision
  - constants 2-5, 9-12, X-6
  - definition of X-9
  - storage allocated A-1
  - variables 2-8, 10-6
- single quotation escape sequence 9-10
- sizeof operator 13-21
- source file 5-4
- space character 1-6, 16-4
- space, addressable
  - internal variables A-4
  - register variables A-4
- space, heap A-6
- space, stack frame A-6
- stack frame A-6
- standard library
  - printf function 4-8
  - scanf function 4-14
- statements 3-6, 15-3
  - block 3-6, 15-4
  - conditional 3-8
    - if 15-20
    - switch 15-28
  - control 3-12
    - break 15-6
    - continue 15-9
    - goto 15-18
    - return 15-26
- expression 3-7, 15-14
- labeled 15-23
- looping 3-10
  - do 15-12
  - for 15-15
  - while 15-34



primary expression 13-11  
return statement 15-26  
statement 15-3  
string constant 9-21  
struct or union specifier 11-31, 11-39  
subdeclarator 8-8  
subscript declarator 11-8  
switch body 15-29  
switch statement 15-28  
type definition 11-4  
unary expression 13-16  
while statement 15-34

## T

tab characters  
    horizontal escape sequence \t 9-10  
    vertical escape sequence \v 9-10  
    white space, as 1-6, 16-4  
ternary expression ? : 13-31  
type balancing 14-6  
type definition 11-4  
type specifier 2-8  
typedef 11-4  
types  
    array 11-8  
    character 10-4  
    enumeration 11-17  
    enumerations 11-17  
    floating-point 10-6  
    functions 12-5  
    IBM RT PC C Language compiler sizes  
        of A-1  
    integer 10-8  
    pointer 11-22  
    structure 11-30  
    union 11-39  
    void 10-10, 14-10

## U

unary conversions, usual 14-4  
unary expression 13-16  
undef preprocessor statement 16-9  
union specifier 11-39  
unions  
    declarations 11-39  
    description of 11-39  
    members of 13-15  
unsigned 10-8  
unsigned arithmetic expansion  
    conversions 14-9  
unsigned char 10-4, A-1  
unsigned int 10-8, A-1  
unsigned long 10-8, A-1  
unsigned short 10-8, A-1

## V

value parameter 4-8  
variables  
    array 11-8  
    character 10-4  
    definition of 2-6  
    enumeration 11-17  
    floating-point 10-6  
    integer 10-8  
    pointer 11-22  
    structure 11-30  
    union 11-39  
vertical tab escape sequence \v 9-10  
void 10-10  
void conversions 14-8, 14-10  
void function 12-11  
volatile 8-9  
volatile conversions 14-8, 14-10

---

**W**

while statement 3-10, 15-34  
white space 1-6, 16-4, 16-6  
widening 14-5  
word, definition of X-11





The IBM RT PC  
Programming Family

### **Reader's Comment Form**

### **IBM RT PC C Language Guide and Reference**

SC23-0803

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

Tape

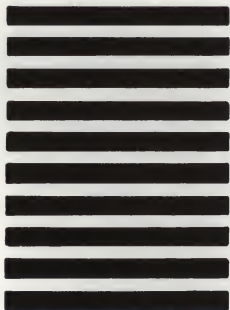
Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and tape

Fold and tape



International Business Machines Corporation  
Department 997, Building 998  
11400 Burnet Rd.  
Austin, Texas 78758

POSTAGE WILL BE PAID BY ADDRESSEE

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

**BUSINESS REPLY MAIL**



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES



The IBM RT PC  
Programming Family

### **Reader's Comment Form**

### **IBM RT PC C Language Guide and Reference**

SC23-0803

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

Tape

Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and tape

Fold and tape

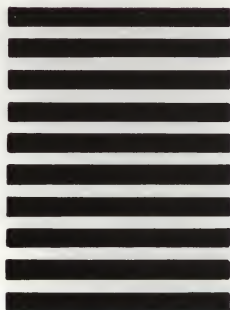
International Business Machines Corporation  
Department 997, Building 998  
11400 Burnet Rd.  
Austin, Texas 78758

POSTAGE WILL BE PAID BY ADDRESSEE

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

**BUSINESS REPLY MAIL**

NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES



Book Title \_\_\_\_\_

Order No. \_\_\_\_\_

**Book Evaluation Form**

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y    N    Is the purpose of this book clear?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is the table of contents helpful?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is the index complete?

\_\_\_\_\_  
\_\_\_\_\_Y    N    Are the chapter titles and other headings  
meaningful?\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is the information organized appropriately?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is the information accurate?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is the information complete?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Is only necessary information included?

\_\_\_\_\_  
\_\_\_\_\_Y    N    Does the book refer you to the appropriate  
places for more information?\_\_\_\_\_  
\_\_\_\_\_

Y    N    Are terms defined clearly?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Are terms used consistently?

\_\_\_\_\_  
\_\_\_\_\_Y    N    Are the abbreviations and acronyms  
understandable?\_\_\_\_\_  
\_\_\_\_\_

Y    N    Are the examples clear?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Are examples provided where they are needed?

\_\_\_\_\_  
\_\_\_\_\_

Y    N    Are the illustrations clear?

\_\_\_\_\_  
\_\_\_\_\_Y    N    Is the format of the book (shape, size, color)  
effective?\_\_\_\_\_  
\_\_\_\_\_**Other Comments**

What could we do to make this book or the entire set of  
books for this system easier to use?

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_**Optional Information**

Your name \_\_\_\_\_

Company name \_\_\_\_\_

Street address \_\_\_\_\_

City, State, ZIP \_\_\_\_\_

Tape

Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and tape

Fold and tape

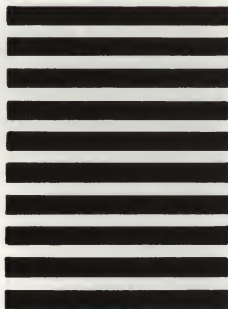
International Business Machines Corporation  
Department 997, Building 998  
11400 Burnet Rd.  
Austin, Texas 78758

POSTAGE WILL BE PAID BY ADDRESSEE

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

**BUSINESS REPLY MAIL**

NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES





© IBM Corp. 1987  
All rights reserved.

International Business  
Machines Corporation  
Department 997, Building 998  
11400 Burnet Rd.  
Austin, Texas 78758

Printed in the  
United States of America

SC23-0803-0



SC23-0803-00



92X1280